Julie McKeehan / Neil Rhodes

PROGRAMMING FOR THE NEWTON®

Software Development with NewtonScript™

Foreword by
Walter R. Smith

Enclosed Disk contains a demonstration version of **Newton Toolkit**™ —Apple Computer's complete development environment for Newton®

AP PROFESSIONAL

Programming for the Newton®

Software Development with NewtonScript™

LIMITED WARRANTY AND DISCLAIMER OF LIABILITY

ACADEMIC PRESS, INC. ("AP") AND ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION OR PRODUCTION OF THE ACCOMPANYING CODE ("THE PRODUCT") CANNOT AND DO NOT WARRANT THE PERFORMANCE OR RESULTS THAT MAY BE OBTAINED BY USING THE PRODUCT. THE PRODUCT IS SOLD "AS IS" WITHOUT WARRANTY OF ANY KIND (EXCEPT AS HEREAFTER DESCRIBED), EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY OF PERFORMANCE OR ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. AP WARRANTS ONLY THAT THE MAGNETIC DISKETTE(S) ON WHICH THE CODE IS RECORDED IS FREE FROM DEFECTS IN MATERIAL AND FAULTY WORKMANSHIP UNDER THE NORMAL USE AND SERVICE FOR A PERIOD OF NINETY (90) DAYS FROM THE DATE THE PRODUCT IS DELIVERED. THE PURCHASER'S SOLE AND EXCLUSIVE REMEDY IN THE EVENT OF A DEFECT IS EXPRESSLY LIMITED TO EITHER REPLACEMENT OF THE DISKETTE(S) OR REFUND OF THE PURCHASE PRICE, AT AP'S SOLE DISCRETION.

IN NO EVENT, WHETHER AS A RESULT OF BREACH OF CONTRACT, WARRANTY OR TORT (INCLUDING NEGLIGENCE), WILL AP OR ANYONE WHO HAS BEEN INVOLVED IN THE CREATION OR PRODUCTION OF THE PRODUCT BE LIABLE TO PURCHASER FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT OR ANY MODIFICATIONS THEREOF, OR DUE TO THE CONTENTS OF THE CODE, EVEN IF AP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

Any request for replacement of a defective diskette must be postage prepaid and must be accompanied by the original defective diskette, your mailing address and telephone number, and proof of date of purchase and purchase price. Send such requests, stating the nature of the problem, to Academic Press Customer Service, 6277 Sea Harbor Drive, Orlando, FL 32887, 1-800-321-5068. APP shall have no obligation to refund the purchase price or to replace a diskette based on claims of defects in the nature or operation of the Product.

Some states do not allow limitation on how long an implied warranty lasts, not exclusions or limitations of incidental or consequential damages, so the above limitations and exclusions may not apply to you. This Warranty gives you specific legal rights, and you may also have other rights which vary from jurisdiction to jurisdiction.

THE RE-EXPORT OF UNITED STATES ORIGIN SOFTWARE IS SUBJECT TO THE UNITED STATES LAWS UNDER THE EXPORT ADMINISTRATION ACT OF 1969 AS AMENDED. ANY FURTHER SALE OF THE PRODUCT SHALL BE IN COMPLIANCE WITH THE UNITED STATES DEPARTMENT OF COMMERCE ADMINISTRATION REGULATIONS. COMPLIANCE WITH SUCH REGULATIONS IS YOUR RESPONSIBILITY AND NOT THE RESPONSIBILITY OF AP.

Programming for the Newton®

Software Development with NewtonScript™

Julie McKeehan and Neil Rhodes

Calliope Enterprises
Redlands, California

AP PROFESSIONAL

Boston San Diego New York
London Sydney Tokyo Toronto

This book is printed on acid-free paper. ♾

AP PROFESSIONAL
955 Massachusetts Avenue, Cambridge, MA 02139

An imprint of ACADEMIC PRESS, INC.
A Division of HARCOURT BRACE & COMPANY

Library of Congress Cataloging-in-Publication Data

McKeehan, Julie.
Programming for the Newton / Julie McKeehan and Neil Rhodes.
p. cm.
Includes bibliographical references and index.
ISBN 0-12-484800-1 (acid-free paper)
1. Computer software--Development. 2. Newton Toolkit.
3. NewtonScript. I. Rhodes, Neil, 1962- . II. Title.
QA76.76.D47M4 1994
005.265--dc20 94-10094
CIP

Printed in the United States of America
94 95 96 97 98 IP 9 8 7 6 5 4 3 2 1

Apple Computer, Inc. Software License

PLEASE READ THIS LICENSE CAREFULLY BEFORE OPENING THE PACKAGE CONTAINING THE SOFTWARE. BY USING THE SOFTWARE YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, PROMPTLY RETURN THE UNUSED SOFTWARE DISK AND BOOK TO THE PLACE WHERE YOU OBTAINED IT AND YOUR MONEY WILL BE REFUNDED.

1. License. The application, demonstration, system and certain other software accompanying this License, whether on disk, in read only memory, or on any other media (the "Apple Software"), the related documentation and fonts are licensed to you by Apple. You own the disk on which the Apple Software and fonts are recorded but Apple and/or Apple's Licensor(s) retain title to the Apple Software, related documentation and fonts. This License allows you to use the Apple Software and fonts on a single Apple computer and make one copy of the Apple Software and fonts in machine-readable form for backup purposes only. You must reproduce on such copy the Apple copyright notice and any other proprietary legends that were on the original copy of the Apple Software and fonts. You may also transfer all your license rights in the Apple Software and fonts, the backup copy of the Apple Software and fonts, the related documentation and a copy of this License to another party, provided the other party reads and agrees to accept the terms and conditions of this License.

2. Restrictions. The Apple Software contains copyrighted material, trade secrets and other proprietary material and in order to protect them you may not decompile, reverse engineer, disassemble or otherwise reduce the Apple Software to a human-perceivable form. You may not modify, network, rent, lease, loan, distribute or create derivative works based upon the Apple Software in whole or in part. You may not electronically transmit the Apple Software from one computer to another or over a network. The use authorized hereunder is expressly limited to use in connection with doing the exercises contained in the book with which it is distributed and does not include the right to use the Apple software for purposes of building applications for any Newton-based products.

3. Termination. This License is effective until terminated. You may terminate this License at any time by destroying the Apple Software, related documentation and fonts and all copies thereof. This License will terminate immediately without notice from Apple if you fail to comply with any provision of this License. Upon termination you must destroy the Apple Software, related documentation and fonts and all copies thereof.

4. Export Law Assurances. You agree and certify that neither the Apple Software nor any other technical data received from Apple, nor the direct product thereof, will be exported outside the United States except as authorized and as permitted by the laws and regulations of the United States. If the Apple Software has been rightfully obtained by you outside of the United States, you agree that you will not re-export the Apple Software nor any other technical data received from Apple, nor the direct product thereof, except as permitted by the laws and regulations of the United States and the laws and regulations of the jurisdiction in which you obtained the Apple Software.

5. Government End Users. If you are acquiring the Apple Software and fonts on behalf of any unit or agency of the United States Government, the following provisions apply. The Government agrees:

(i) if the Apple Software and fonts are supplied to the Department of Defense (DoD), the Apple Software and fonts are classified as "Commercial Computer Software" and the Government is acquiring only "restricted rights" in the Apple Software, its documentation and fonts as that term is defined in Clause 252.227-7013(c)(1) of the DFARS; and

(ii) if the Apple Software and fonts are supplied to any unit or agency of the United States Government other than DoD, the Government's rights in the Apple Software, its documentation and fonts will be as defined in Clause 52.227-19(c)(2) of the FAR or, in the case of NASA, in clause 18-52.227-86(d) of the NASA Supplement to the FAR.

6. Limited Warranty on Media. Apple warrants the diskettes and/or compact disc on which the Apple Software and fonts are recorded to be free from defects in material and workmanship under normal use for a period of ninety (90) days

from the date of purchase as evidenced by a copy of the receipt. Apple's entire liability and your exclusive remedy will be replacement of the diskettes and/or compact disc not meeting Apple's limited warranty and which is returned to Apple or an Apple authorized representative with a copy of the receipt. Apple will have no responsibility to replace a disk/disc damaged by accident, abuse or misapplication. ANY IMPLIED WARRANTIES ON THE DISKETTES AND/OR COMPACT DISC, INCLUDING THE IMPLIED WARRANTIES OF MERCHNATABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE LIMITED IN DURATION TO NINETY (90) DAYS FORM THE DATE OF DELIVERY. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY BY JURISDICTION.

7. Disclaimer of Warranty on Apple Software. You expressly acknowledge and agree that use of the Apple Software and fonts is at your sole risk. The Apple Software, related documentation and fonts are provided "AS IS" and without warranty of any kind and Apple and Apple's Licensor(s) (for the purposes of provisions 7 and 8, Apple and Apple's Licensor(s) shall be collectively referred to as "Apple") EXPRESSLY DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. APPLE DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE APPLE SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE APPLE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE APPLE SOFTWARE AND THE FONTS WILL BE CORRECTED. FURTHERMORE, APPLE DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE APPLE SOFTWARE AND FONTS OR RELATED DOCUMENTATION IN TERMS OF THEIR CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY. SHOULD THE APPLE SOFTWARE PROVE DEFECTIVE, YOU (AND NOT APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU.

8. Limitation of Liability. UNDER NO CIRCUMSTANCES INCLUDING NEGLIGENCE, SHALL APPLE BE LIABLE FOR ANY INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES THAT RESULT FROM THE USE OR INABILITY TO USE THE APPLE SOFTWARE OR RELATED DOCUMENTATION, EVEN IF APPLE OR AN APPLE AUTHORIZED REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

In no event shall Apple's total liability to you for all damages, losses, and causes of action (whether in contract, tort (including negligence) or otherwise) exceed the amount paid by you for the Apple Software and fonts.

9. Controlling Law and Severability. This License shall be governed by and construed in accordance with the laws of the United States and the State of California, as applied to agreements entered into and to be performed entirely within California between California residents. If for any reason a court of competent jurisdiction finds any provision of this License, or portion thereof, to be unenforceable, that provision of the License shall be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this license shall continue in full force and effect.

10. Complete Agreement. This License constitutes the entire agreement between the parties with respect to the use of the Apple Software, related documentation and fonts, and supersedes all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by a duly authorized representative of Apple.

Contents

Foreword by Walter Smith

If Newton is your first programming platform, congratulations: you won't have to unlearn as much. The rest of us are so used to the archaic baggage attached to traditional development systems that Newton can be a little confusing, just because of all the things you don't have to do to develop for it.

Here's a typical scene you may find familiar. The first thing you do when unpacking a new development system is find the C compiler and try to get it to work. Once you get it to compile "hello world", you start looking for clues on how to get a window up on the screen. After locating the header file you need, trashing memory and rebooting your machine a few times by failing to initialize a pointer or two, and typing in a surprising amount of code, the window appears. Now you can start to think about how you might resize it. Next week, maybe, you might be getting around to what you actually wanted to do.

Now consider the Newton experience. You sit down with Newton Toolkit and draw most of your application's user interface, including pictures, buttons, pop-up menus, and type-specific data fields. You download the application to a Newton and try it out to see if you like the feel. Once the interface looks good, you start to put some code under it, a piece at a time. Perhaps you save some time by dropping in some interface elements you've already used in your other Newton applications. Newton makes it easy to progressively prototype and try your application so you know you're headed in the right direction.

You'll do a lot without actually writing any code. (In fact, for the first 100 pages of this book, there isn't any code at all.) When you do write some, it will be in NewtonScript. If you're accustomed to a low-level language like C++, NewtonScript will take some getting used to. You can't trash memory with dangling pointers, because there's no way to make one. You can't forget to dispose memory, or accidentally dispose it twice, because the system disposes it for you. You can make a specialized object without having to invent a new class just for it, because you don't need classes. The list goes on, as you'll see.

One of the first things you'll notice in Newton Toolkit is a huge palette of "view templates"—user-interface elements that are preprogrammed and waiting for you in every Newton device. They are fully functional: you don't have to tell the pop-up menu how to pop up. You just determine the behavior specific to your

application. That means less code for you to write, less of your effort wasted, and a smaller delivered application.

Of course, it wouldn't matter how easy it was to develop for Newton unless the platform itself had something going for it. Newton is the leading platform for a new product category, with built-in features that include communications, intelligent assistance, handwriting and graphics recognition, persistent object storage with desktop data synchronization, and integrated e-mail, printing, and faxing. It all works on small, light, low-power hardware targeted at the professional and consumer markets.

Newton's future is bright because of the excitement and momentum in the third-party community. Apple is licensing the hardware and software technology widely, and the licensees are developing their own Newton devices with all kinds of capabilities. The dynamic nature of NewtonScript will make it easy for you to take advantage of these special features. Newton applications are inherently portable, which means your software can run on all of these products—even with different microprocessors.

The Newton world is young. If you have great ideas for Newton products, there's room for you to make them a reality. Fire up your Newton Toolkit and help us define the future.

Walter Smith
Newton Group
Personal Interactive Electronics Division
Apple Computer, Inc.
February 1994

Acknowledgements

We actually started this book long before we began typing the words. We spent the last half of 1993 working on the Newton programming class for Apple. It was an exciting time to be at Apple and inspiring to see the dedication of many fine people who were doing two things simultaneously: launch Newton and ensure that Newton developers would have the support services they needed. All of the people at PIE Developer Training and PIE Developer Technical Support often went without sleep to see that training materials were just a little bit more accurate, that one more crucial question was answered, and one more developer helped. Most of all, they ensured that we were properly trained so that we could in turn teach you.

Most especially our thanks to Gabriel Acosta-Lopez, a supremely great guy. He is the lone ranger of Developer Training at PIE, and a fine one to have. Gabriel's good company, helpful advice, and conscientious attitude make him a pleasure to work with—even when he is not being funny. Also, we want to thank Steve Strong for his long friendship and support of our work. His administrator's job often involves long and grim hours without many immediate tangible benefits; this all to ensure that Newton developers get enough support from Apple.

We particularly want to acknowledge Mike Engbar, Kent Sanvick, ZZ Zimmerman, Bob Ebert, and Maurice Sharp of PIE Developer Technical Support—the bug guys. They have gone over our training materials for errors more times that we can remember. Mike, Kent, and Walter Smith also reviewed this manuscript for errors that might have crept in—catching quite a few we might add. It is great to have the generous assistance of such fine people when you are going to lay it on the line in a book about new technology.

Then there are our students at Apple Developer University. They were our guinea pigs for the development of the programming course and ultimately for this book. From them, we learned what worked, what needed more time, and what a bright future Newton has. The folks at Apple may have seen to it that we understood Newton correctly; but our students patiently helped us refine the training materials, and in the long run made this book all the better for it.

On the book end of things, there are many people to whom we owe a debt of gratitude. We particularly thank our editor, Charles Glaser, at AP Professional.

He saw the potential in Newton at its introduction at MacWorld and had the courage to stand behind his vision ever since. An author could have no better editor. He had the tenacious spirit required to plunge into the stormy waters of Apple Marketing and Legal and emerge successfully with a demonstration version of the Newton development environment. This was not a simple task. We want to thank all the other people of AP Professional for their work in copy editing and promotion, as well. There is also our agent, Carole McClendon, whose skill and professionalism found us this publisher.

We also had two testers, Allan Hoeltje and Matthew Krawitz, who went through the application in the book to ensure that our directions were consistent and clear. Because of their diligent efforts, we were able to catch a lot of problems. They have our gratitude. WaiterHelper was the inspiration of Loring and Barbie Fiske-Phillips; they have our thanks for a great application idea.

Even though these and many other unmentioned people gave time and energy to see to it that this book is without mistake, some no doubt still remain. While the presence of such annoying errors is solely our responsibility, we would be grateful if you report any you find in the text or code. If you, our clever readers, find some new ones, we will also gratefully acknowledge you in the next edition.

Last of all, we have some personal acknowledgments. This book could not have happened without the unending support of people in our day-to-day lives. We most especially want to thank Jean and Peter Drinkwine, Barbie and Loring Fiske-Phillips, and Gloria McKeehan for the kind willingness to handle the unending minutiae of our daily lives. If they had not managed these responsibilities, this book would have taken much longer to write and captured our senses of humor along the way. Our heartfelt and loving gratitude to you.

—Julie McKeehan and Neil Rhodes
March 1994

Preface

The 1980's were a very good decade for computing. The personal computer came into its own and anything was possible. Anyone could have a good idea, work hard, and create some software that would make them famous. Out of this time of possibilities, companies such as Apple and Microsoft grew to prominence.

Gradually over the decade, however, the time required to create an application lengthened from months to years, and the cost to bring a product to market went from nothing to the GNP of many small countries. To complicate matters further, large companies started wanting software customized to their own particular needs even as commercial software applications quickly multiplied.

Enter the Newton in 1993 and once again the time is right—anything is possible. Anyone can have a good idea, work hard, and create a good Newton application in a few months. Remember also that the playing field is still quite empty. Further, the Newton development environment makes customizing software a snap. Even tiny companies can have software tailored especially to their needs.

So, "Why Newton?" you might ask. Because Newton is part of our future, and the direction in which computing will move. Wireless technology will reshape the world, just as the personal computer did before it.

Try and imagine what your software and a Newton could accomplish: helping a doctor make rounds in a hospital; salespeople using the company's current online catalog to take an order and then faxing it back to the office; stock market traders beaming their buy and sell orders across the floor; a Japanese tourist getting the maps and language guides in Kanji for a trip to Disneyland; and everyone telling the VCR what television shows to tape. Whatever your imagination can devise for this wireless world, you can create on the Newton.

It is at least worth investigating. And this book gives you everything you need to do that. There is a floppy disk at the back that contains:

- **A full featured demonstration version of Newton Toolkit, the development environment for the Newton.**
- **The sample application that we create within the book.**

So, why not investigate the future with us.

How to Read This Book

You should first read Appendix G, Using Newton Toolkit. This will familiarize you with the development environment before you begin using it. Open Newton Toolkit and play around with it while you read though this section.

Once you start writing code, you should read Chapter 9, Debugging Your Application. Also browse through the Appendices as they contain the NewtonScript methods, messages, functions, and variables that you will be using in your code.

The Structure of The Book

We have ordered the chapters to coincide with the order in which you create an application. If you have no familiarity with Newton programming, read the book in the order it was written. If you already understand the topics covered in a chapter, feel free to skip it.

The Structure of Each Chapter

Most chapters have been divided into two sections:

- the topic discussion
- the application implementation

The first section is always a topic discussion and will contain a variety of examples to help illustrate those issues. There is also a second section in many chapters where we create the sample application step by step. Feel free to read the topics first and the implementations second if you prefer.

What You Need to Program the Newton

Here is what you need:

- A Newton.
- A Macintosh (a Windows version of Newton Toolkit is under development).

- A serial cable to hook your Newton to the Macintosh. Use a Macintosh/Imagewriter II cable, available from an Apple dealer as model M0197—part number 590-0552-A.

What You Need to Know to Program the Newton

Macintosh Knowledge

You need know how to navigate on the Macintosh desktop (for example, open files, change directories, etc.) but not much beyond that.

Programming Experience

This book expects that you already know how to program. It makes no effort to explain standard programming concepts such as parameters, functions, or variables. We do explain certain object-oriented concepts, such as inheritance.

About the Demonstration Version of Newton Toolkit

The demonstration version of Newton Toolkit that is included with this book is a full featured version that contains some restrictions. You may not distribute any software you produce with it. In order to sell or distribute Newton software, you must buy Newton Toolkit from Apple Computer. This demonstration version is provided so that you can learn Newton programming. Make sure to read the license agreement that accompanies this demonstration NTK.

You should know that Apple is committed to the Newton and to Newton developers. It has created StarCore, a publisher and distributor of Newton software to make sure that your software reaches Newton customers.

Chapter 1

Newton Application Design

The power to make buildings beautiful lies in each of us already.

—Christopher Alexander, The Timeless Way of Building

Overview

You have, perhaps, seen the "What is Newton?" commercial that has issued forth from the hallowed halls of Apple marketing. Just as they are using this metaphysical jingle to educate potential customers about Newton's ability to illumine everyone's lives, so this chapter will educate you about the role of the application and

you, its designer. Only application interface and design are covered in detail; the life cycle of an application is a sweeping overview. It is our hope that by the end of the chapter you will have a good understanding of three things:

- Important Newton terminology.
- The life cycle of a Newton application, from installation to its removal. You will also learn a little about the relationship between the runtime application and the code you create in Newton Toolkit. Likewise, we will touch upon how Newton handles data.
- How to design a Newton application.

After covering the ins and outs of Newton application design we will introduce you to WaiterHelper, the application we create in this book. As you will see, WaiterHelper is an application for waiters taking orders at the Chez Calliope Restaurant and one of the tools we use to teach you Newton programming.

A Newton Glossary

Newton and Newton programming bring new terminology to the playing field. Below is a list of the most important new terms with a brief description of each:

PDA	A personal digital assistant. The Newton is one (Apple would say only) of this new type of device.
MessagePad/ExpertPad	The first twins in a family of products.
MessagePad110	Third family member, with a smaller screen and more memory.
Newton Toolkit	The development environment for Newton applications (also called NTK). There is a version available now for the Macintosh. A Windows™ version is due out in the future. Note that Newton development does not take place on the Newton itself.
NewtonScript	The development language for Newton applications. This is an object oriented, dynamic language.

viewClass	The most primitive view type. All views and protos are ultimately based upon a viewClass, though it may be far back in the ancestor chain.
Views	An object created by the Newton view system that is based upon a template (or occasionally a proto). These exist only on the Newton and are the run-time instantiations of what you create in NTK.
Templates	These are the blueprints of a view that you create in NTK. Templates define both the cosmetic aspects of views and the methods within the application.
Protos	These are blueprints for templates (occasionally for views) and are the elements in your NewtonScript code libraries. There are two kinds of protos: system protos— built into the ROM; and user protos—custom ones you create.
Frame	The most important data structure. A frame is a dynamic collection of named slots. All objects in NewtonScript are frames.
Slot	One element in a frame, composed of a name and a value. A slot can hold any kind of value.
Method	A member function of an object located in a slot of that frame. When an object receives a message, it executes a method.
Message	A request to carry out an action—it is sent to a particular object.
Store	The physical storage device for data. There are currently two for the Newton: internal memory and a PCMCIA card.

Soup	A related collection of data (as in the "Names" soup or the "Calendar" soup). All readable and writable data are stored in soups.
Entry	This is an individual item in a soup (for instance, one name card in the Names soup). An entry is a NewtonScript data frame.
Frame Heap	The Newton RAM reserved for memory that is allocated dynamically by NewtonScript.
InstallScript	An application function that is run when the application is installed.
RemoveScript	An application function that is run when the application is de-installed.
Base view	The main view of an application (at the top of the parent hierarchy). Methods and variables used by the entire application are located here.

The Life Cycle of an Application

So what actually happens when a user puts an application on a Newton (either in memory or on a PCMCIA card)? Let us find out.

Before Installation

Our example is an application that is on a PCMCIA card (though an application installed in memory would act similarly).

First, look at the state of the Newton before inserting the card (see Figure 1.1) There are some applications that are already in the internal memory:

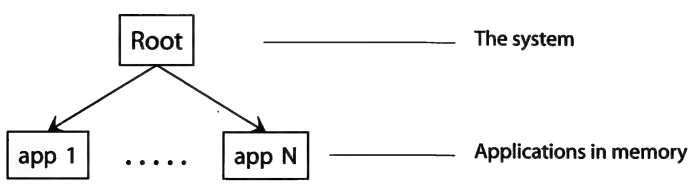


Figure 1.1 State of Newton before inserting a card.

The Sample Application

We will be adding the simple application shown in Figure 1.2. The base view is a plain white slip view that floats above other views; this view also contains two child views. Each child view contains text. The first child view contains, "Enter Name," and the other has the text, "Hank".

The left hand side of Figure 1.2 shows the three views as they are displayed upon the Newton. The right hand side shows the view frames in the Newton Application Memory Heap (frame heap) that underlie what you see on the Newton.

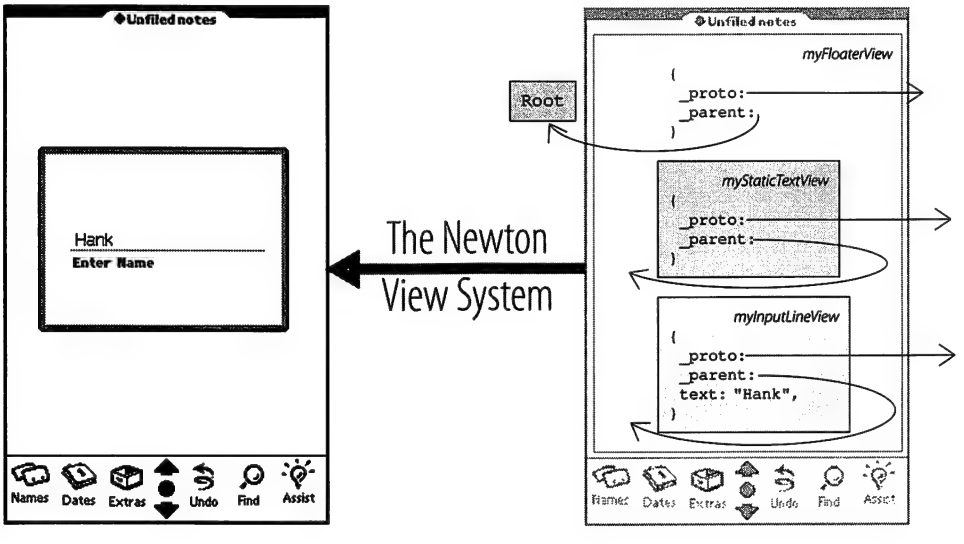


Figure 1.2 An application and the views that underlie it.

Installing an Application

The user pops the card into the Newton. The system updates the application memory heap to include the new application that is located on the card. The New App frame in memory also has a pointer to its template. Templates contain the actual code and data of the applications (see Figure 1.3):

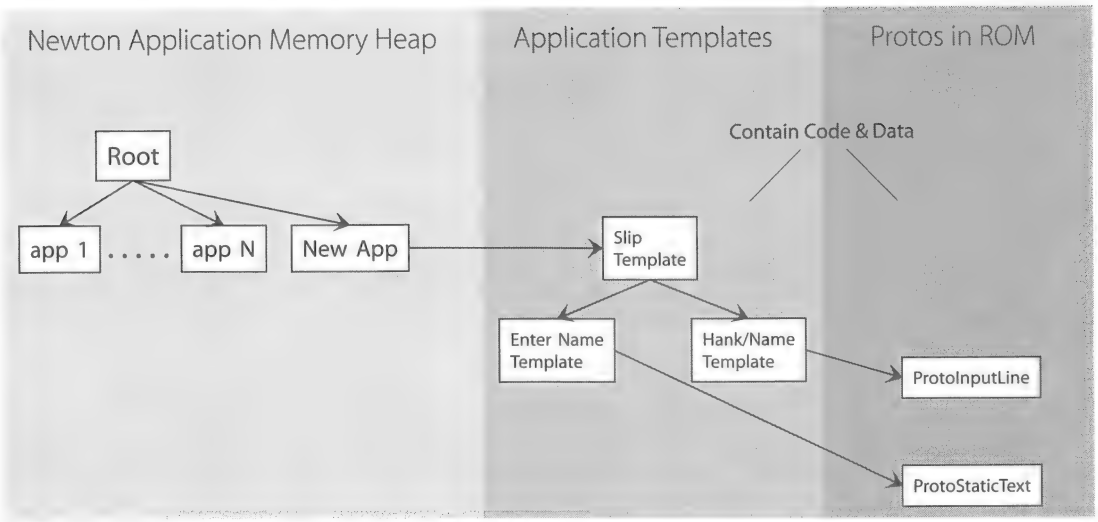


Figure 1.3 Installing an Newton application.

Running an Application

When the user opens an application, the system creates a view for each template. Thus, our simple application opens and has a new set of pointers as shown in Figure 1.4. The child text views now point to their own templates.

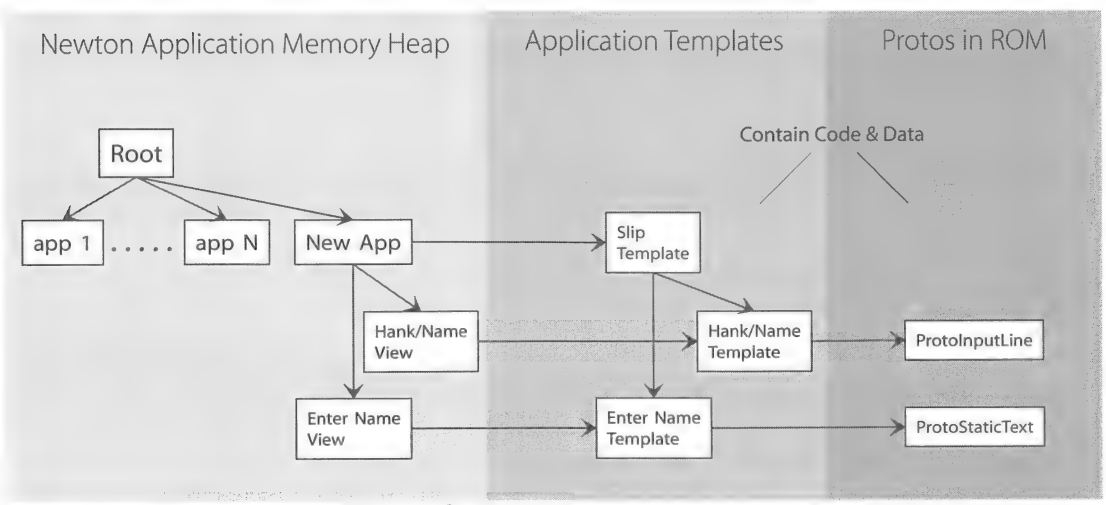


Figure 1.4 An application after it is open.

Closing an Application

There are several ways to close applications: you can tap the close box or retap the application icon in the Extras drawer. When the user closes an application, the system sends a close message to the application's base view. All of the various views are closed, the pointers to templates as disposed of, and you end up with this state of things (see Figure 1.5):

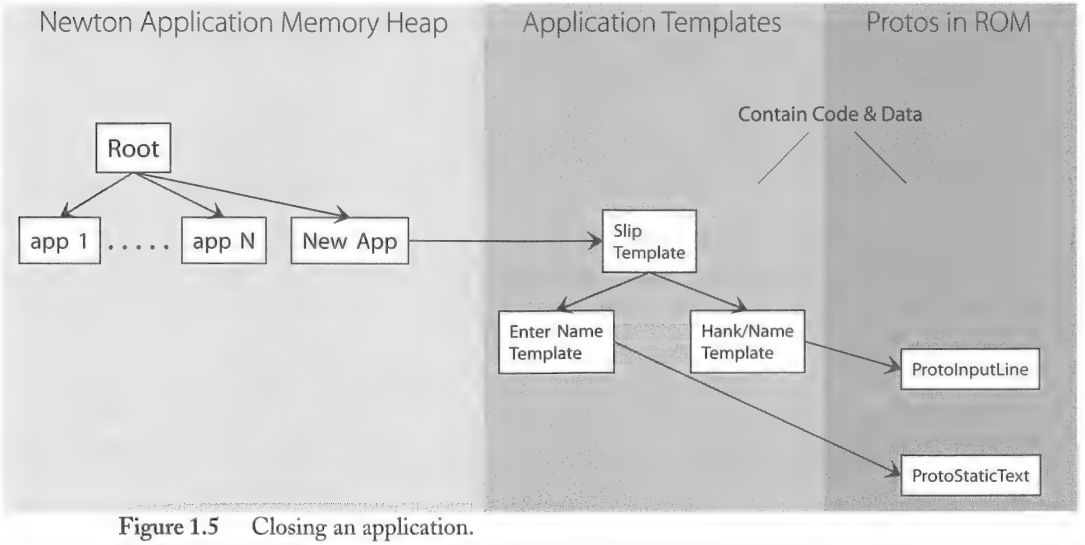


Figure 1.5 Closing an application.

Removing an Application

If the application resides on the PCMCIA card, removing it is pretty simple–pop the card. If the application resides in memory, you need to remove it via Preferences. Once the card is removed, you are back to the state of things in Figure 1.6.

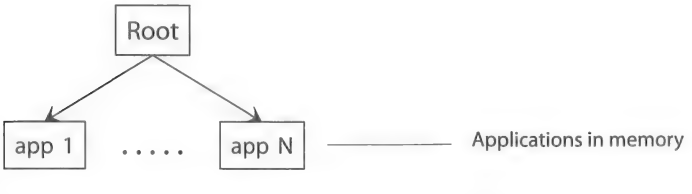


Figure 1.6 Removing an Application from the Newton.

Persistent Data on the Newton

The remaining piece of the application puzzle is Newton's treatment of data. Rather than storing data in a flat file format, Newton stores data using what are called persistent objects. That is, rather than writing a representation of the data structure to a file, the data structure itself is stored. (Thus, the structure is said to persist.) These persistent objects are stored in soups. A single soup contains a collection of related frames. For example, the Names application has a Names soup, which contains all the name entries. One entry in that soup would be an individual name card. Soups can also exist on multiple stores; you could have a Names soup both on the Newton and PCMCIA card.

Sharing data

On many personal computers, information between applications is not shared. A typical example is names and phone numbers. A personal computer user has to enter names and phone numbers innumerable times. First, in a faxing utility, again in an address book, again in a word processor, and so on.

On Newton, a user should never have to repeat such information. Because soups are accessible to any application, users have access to data from anywhere. If you are creating an application that deals with names and phone numbers, you use the built in Names soup to get that data.

In conjunction with this, application writers are encouraged to publish the format of their soup entries so that other applications may access them. Apple also publishes the format of all its soups so that you can obtain information, as well as modify or augment existing information (like adding favorite color to each name in the Names soup).

Newton Interface Design

Now that have a general idea of a Newton application, let us turn to the more abstract issues of good user interface design. There are several helpful guidelines that we will give you. The most important principal for you to remember, however, and unfortunately the hardest to actually put into practice is this:

- *the Newton is not a personal computer*

Do Not Design for a Personal Computer

The Personal Computer World

The world of personal computers is populated with hardware accompanied by mice for movement, large screens, lots of wires, vast amounts of memory, and keyboards as the primary input device. The software is large, complex, and contains vast amounts of redundancy from application to application; for example, word processors have draw tools, draw programs have word processors, and so on.

There is a perception among personal computer application designers that for a software application to be successful it must offer not only a few innovative features, but every other feature found in similar applications (a perception fueled by the press with their bullet chart reviews, no doubt). This has resulted in software that can help with a variety of tasks, but requires extensive training to use.

The Newton World

The Newton world has different users and new tasks. Newton hardware is different as well; it uses:

- The stylus for both movement and input.
- Few wires now and even fewer in the future.
- A variable screen size. It is currently small, but could be anything from a wristwatch to a whiteboard in size.
- Small amounts of memory. This makes it the most precious commodity on the Newton (though eventually this will change).

Obviously each of these hardware issues drives the design process. This, accompanied by the consumer component of the Newton user base, means that applications should be small, doing one thing very well, and other things not at all. Because data is shared on the Newton, you should rely upon other applications to handle other tasks. Do your one task, with such ease of use and style, that you take the user's breath away.

A successful Newton application will also combine the right interface elements from the pen and paper world and from the computer world. Further, because the Newton is basically wireless, your users will be on the move and not sitting quietly at a desk; too much sun and bumpy airplane rides are something your design must accommodate.

Since the Newton's hardware is different, its software adaptable to a greater range of enviornments, and application goals more straightforward, you must avoid the beguiling temptation to simply port a scaled down application, design and all, from another platform to the Newton. This is true even if the platform is Macintosh. A Newton application is not just a smaller version of a Macintosh application; do not fill it with Macintosh scroll arrows, look alike windows and tool palettes.

General Design Guidelines

Now that you have heard the ideological lecture, let us discuss more tangilble guidelines. We will start off with the more general issues and end with specific suggestions.

Keep it Small

As we said previously, avoid creeping featurism. Users will not want your application if it is not small enough to easily fit on either the Newton or a card. Nor is it acceptable to require a 2 MB card for each application (unless you plan to give the user a card). Cards are expensive and users will shy away from applications that take up a lot of space. Further, attempting to fit large applications in memory or on small cards is very frustrating. The moral should be clear: keep your applications small, ideally under 200K. Unless you want crabby customers, do not create 1 MB applications.

Keep it Simple

When we say, "do one task extremely well," we obviously do not mean that the task has to be excessively narrow in definition. Rather, we mean avoid adding features that do not directly help with the task; this just adds redundancy. In defining your application, be clear about a particular task a user has to perform and provide an application that makes this easier. Perhaps, two examples will help to clarify this notion of simplicity. Here are two possible Newton applications that interact with each other (on a personal computer this would be one application).

A cookbook—This application helps you figure out what food to cook. Perhaps it is a cookbook based upon the bestselling culinary delights of Jeff Smith, the Frugal Gourmet. You can select recipes in the book and get a set of menus created for you. The application also deals with preparation times and figures out the right

order to do tasks and when to start. If requested, it also gives you a list of required ingredients.

Grocery Store Helper–This application helps you shop at the grocery store. You could get the ingredient list you already created with the cookbook and add other items you need. The application reorders items to match the order you will encounter them in the store (it figures this out by the order that things are checked off), it does price and size comparisons, provides easy checkoff operations, and so on.

Keep it Personal

The Newton in its current implementation is a very personal machine; it is meant to belong to one person. Unlike the desktop computer which can be used by many people, the Newton trains to one person's writing and is used fairly exclusively.

This means that your application should strive to be personal as well. Where applicable, let users personalize the application, better yet do it for them. For example, make views moveable and remember their locations.

Remember that one of the greatest opportunities in this marketplace is in creating custom applications built for particular individuals or companies. NTK's quick turn around time for development makes this a snap. For example, you could write a variation of WaiterHelper exclusively for Alice's Restaurant, or an inventory control program based solely for the stock of Acme's mail order warehouse.

Newton Applications Must Be Easy to Use

Admittedly, making an application easy to use is quite difficult. Nevertheless, it is on this point that your application will stand or fall. It is worth your time to think through your design for its ease of use. This is especially true when you realize that a great number of Newton users are not computer users. Thus, the first point of advice:

- Have a *non-computer* user who is skilled in the application task area test your application.

For example: waiters and restaurant owners who do not user computers would be the logical testers of WaiterHelper; cooks the logical testers of a cookBook. While this advice is somewhat applicable for other platforms as well, it is crucial for the Newton, whose user base contains a greater mix of consumers who are not computer users.

NTK also makes the iterative process of application development far easier and quicker than traditional development environments. So, take advantage of it to implement the next piece of advice:

- Refine your application's interface step-by-step.

Emphasize Tapping over Writing

On Newton, users input everything with a stylus (or a fingernail in a pinch). They can write words or tap on things. Actually, you could literally make everything the user sees tappable (a good example of a fully tappable application is an interactive book). This raises an important point for you to remember:

- It is always easier to tap than to write.

Compare an application which emphasizes writing (like the built-in Names application) with one which emphasizes tapping (like WaiterHelper), and you will understand the point. Tapping is one of the most intuitive gestures on the Newton, so use it as much as possible. It is always to be preferred above writing.

Prefer Pickers Over Input Lines

Whenever possible put items in pickers. Figure 1.7 shows an example of three different ways to handle a form of address in an application. One way is to simply have the user write the word in the input line. A better way is to supply a predefined list of common forms, then a name could be picked or written. Better yet is to supply your users with dynamic pickers. The user can either pick from the list, or write a word, or have the additional option of having new words added to the picker and thereafter available in the list.

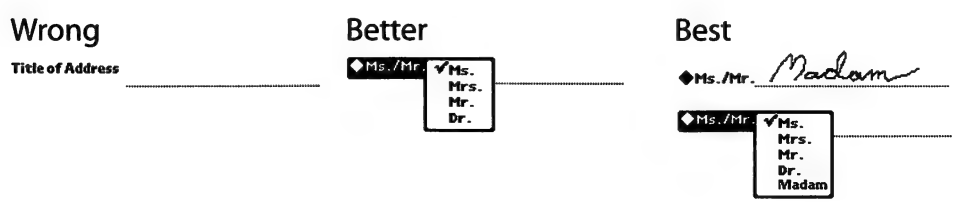


Figure 1.7 Three possible ways to handle form of address input.

Obviously, you could take this rule to an idiotic extreme. In many cases, the user is entering text for which there is no effective limit (you do not know what they will enter). Where is the dividing line? The best way to solve this problem is to look at several sample applications and use common sense. For example, cities are probably too plentiful to put in a picker, but countries might not be. You might also provide a picker with a set number of entries and let the user personalize those.

Anticipate the user's action

Offering dynamic pickers is obviously one way of anticipating user actions, but there are several other ways as well. The interface rule is simple:

- Whereever possible, anticipate what the user wants to do and go ahead and do it yourself.

Here are some examples:

auto number items	For example, in a checkbook application, you should auto number the checks.
preset items	For example, in applications that use the date, set the entry to the current date when new items are created.
pre-fill known entries	For example, if a user does an Intelligent assistance of "Pay Bob" then fill out a new check to the last found Bob in a checkbook application.

Give the User Update Information

You need to give users two important types of update information:

- status information to let users know what is happening, particularly when an action takes too long
- sound or visual effects to show when a view or button has been changed

Status Information

When an action may take more than two or three seconds, you need to give the user some kind of update information. This status information will usually take the form of a slip indicating what is happening (see Figure 1.8).

Figure 1.8 A status slip that updates the user on what is happening.

Another way to give the user information is to invert a view, icon or button, when it is tapped upon, and keep it that way until the action is completed. This tells the user that something is occurring and they need to be patient for a moment. Do not rely upon this for things that take longer than about 5 seconds.

Without this status information, users get confused and assume nothing is happening. They will frequently tap items again, thereby exacerbating the problem further. The end result in such cases will be frustrated users who hold your application responsible.

View and Button Effects

The particular view and button effects you use are somewhat less important than providing them in the first place. The rule here is simple:

- Users need visual and/or auditory feedback to let them know when events are occurring.

For example, when the user taps a button it should be highlighted and perhaps click. Likewise, when the scroll buttons are used, the view being scrolled should not only scroll, but have some obvious visual effect to indicate the change (changing only the view's contents is usually too subtle). There are many different kinds of effects that you can add, from shuffling to zooming in or out. Thankfully, NTK and NewtonScript make the process of adding visual and auditory effects to views extremely easy.

Newton Applications Should Be Consistent

Certain interface items should be placed consistently in application after application. These items include the general placement of buttons, displays, and user input areas. Let us look at the built-in Dates application and another application as examples (see Figure 1.9).

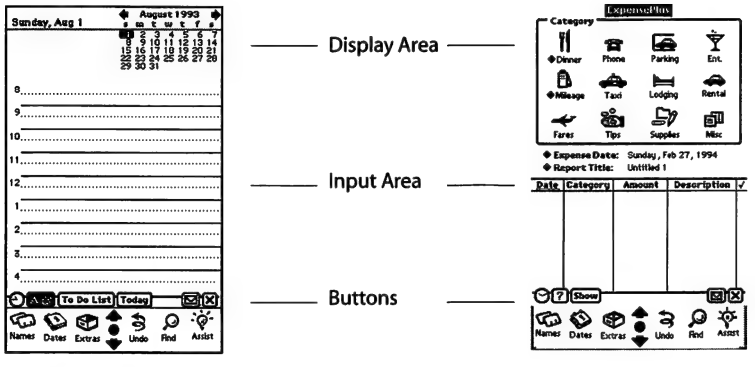


Figure 1.9 Example Applications.

Displays

As you can tell from Figure 1.9, the displays in the Dates are located at the top. The rationale for this is simple:

- A display that needs to be seen when entering information should be located at the top of the screen; otherwise, the user's hand will obscure it.

The built-in Date application is a good example of this. Users will frequently need to see the actual date when entering a meeting. Thus, the calendar is above the scheduling area where the hand is writing (see Figure 1.9).

User Input Area

The placement rule for user input areas, while straightforward, is intimately tied to the presence of displays:

- Place input areas immediately below display information–usually in the middle of the screen.

Such placement keeps displays accessible while leaving as much space for input as possible. If there are no displays, (for example, as in the Names application) feel free to start input areas at the top.

Buttons

Button placement is somewhat more complicated as it depends upon context. Typically, you will place buttons is one of two locations (see Figure 1.10):

- In the status bar at the bottom of the application.
- Within an internal view, in a clear and distinguishable order.

The standard buttons should be placed in particular locations (see Figure 1.10) that users will come to recognize:

Clock	Leftmost button on Status bar
Close	Rightmost button on Status bar
Action	Next to Close button on right hand side
Filing	Next to Routing button on right hand side

You should use the built-in applications as a guide for the placement of most other types of standard buttons (for example, those found in slips). Also, remember to have at least 3 to 4 pixels of space between buttons and make sure that custom buttons are clearly attached to the items they effect.

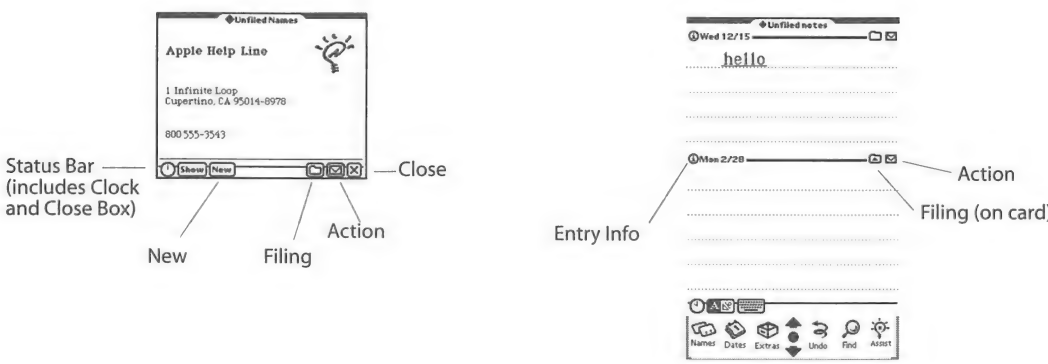


Figure 1.10 Buttons in an application.

Use the Built-in Applications as an Interface Guide

The built-in applications can offer you some valuable design hints. The original built-in applications are not, however, perfect implementations. They were written while the hardware was being created, without the opportunity for testing with handheld Newtons.

In the meantime, do not be afraid to implement something different than you find in the built-in applications. We have already shown you a better way to handle address titles than the Names application uses, there are likewise other areas where your application would be all the better for a different interface.

Do Not Underestimate the Value of Wiz Bang Features

Instead of adding more features to your application, make the ones you have elegant, easy to use, and fun (a smiling user is a great product advertisement). As we have said, you should add sound and view effects as status indicators for your user. Likewise, animation can contribute greatly to a pleasurable application experience.

For example, the system provides a Delete routine that gives you some nice animation for no extra work. To see an example of Delete, simply delete a name card from the Names application. As you can see, the item is crumbled into a ball, and is thrown into a suddenly appearing trash can that then disappears. Users find such animations a pleasure to watch. (We think there is a good third party opportunity in writing an Undo function that makes the trashcan reappear, uncrumples the item, leaving a wrinkle or two, and then makes the can disappear.)

When to Leave Partial Views

Everything that is visible on the screen can potentially be tapped upon by the user. This makes for a clear interface rule:

- If you do not want the user to tap on another application or view, do not let it show.

A good example of this is the Note pad, which is always active. If your application does not use up the entire screen, the user can write on the part of the note pad that is shown.

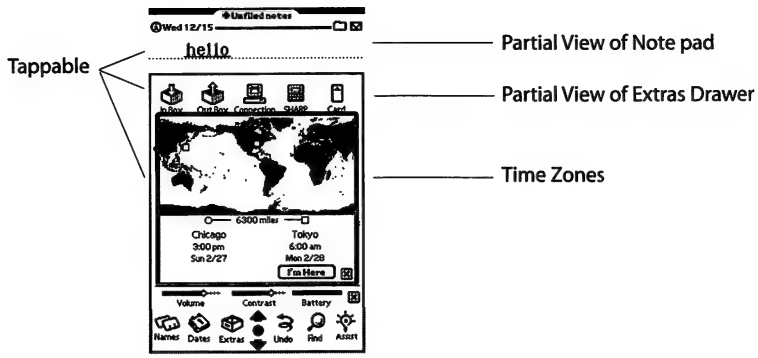


Figure 1.11 Two partial views showing behind Time Zones.

Use the Correct Proto or viewClass

Within the NTK tool palette, there are many choices for a proto to use as a templates base (see "The system protos." on page 95). In some cases, it is obvious which proto to select. For example, when you want a non-tappable label for an item you will probably use a protoStaticText. In many cases, it is not as clear which proto or viewClass you should use for the job. We have already said that some judgement is required in whether to use a protoLabelInputLine (a picker list) or a protoInputLine.

You should become familiar with all of the characteristics of the viewClasses and protos so that you will know which will best fit the design you have in mind. Do not forget that the right alternative may be to design your own proto to handle the task (something we will do later, see "Creating and Using User Protos" on page 98).

Newton Application Designs

Let us look at three commercial Newton applications to see how they have implemented these design guidelines:

- Newton-like in look and feel
- Small
- Simple

- Personal
- Easy to Use
- Emphasize tapping over writing
- Use pickers instead of input lines
- Anticipate the user's action
- Give status information
- Provide visual and auditory effects
- Place displays at the top, input areas in the middle, and buttons at the bottom
- Add some wiz bang to their features

Remember, these are not application reviews, but rather small glimpses of one particular aspect of each application: their interface designs. The first application is ExpensePlus™ (version 1.01), a business expense reporting application, developed by State of the Art, Inc. Figure 1.12 contains some screen dumps from the application and highlights of its design.

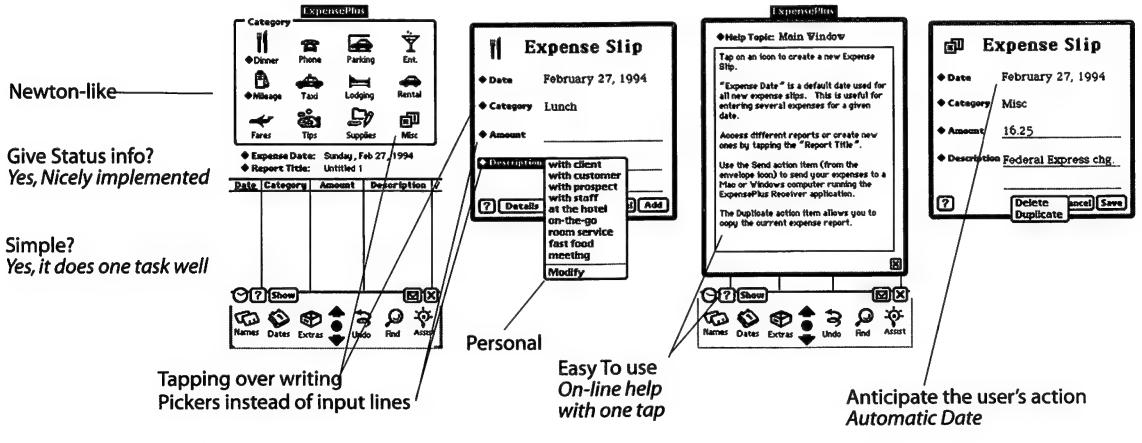


Figure 1.12 ExpensePlus™—an accounting application for the Newton.

The next application is also an expense application. This one is DayTimer® Expense Assistant (version 1.0) by Slate Corporation (see Figure 1.13).

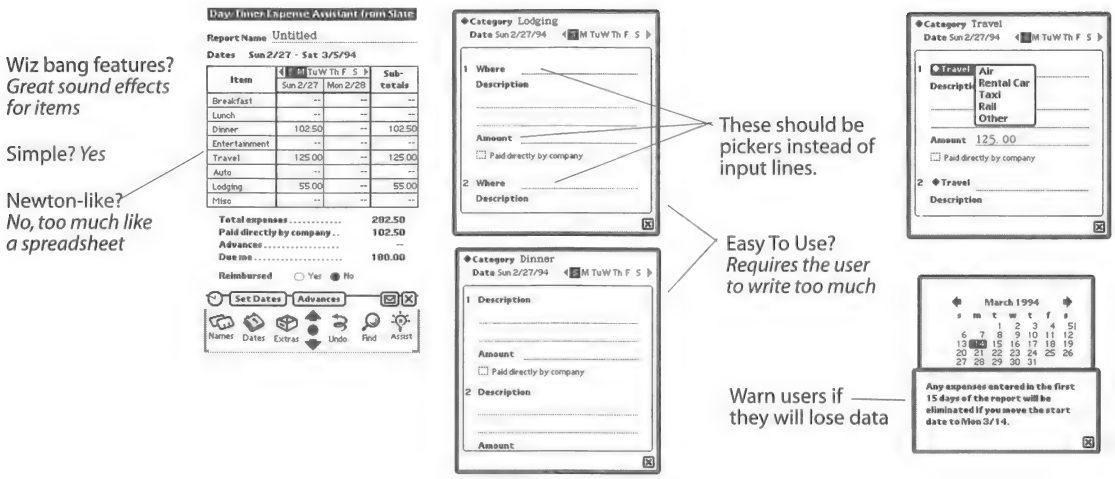


Figure 1.13 DayTimer®Expense Assistant and Meeting Assistant.

The next application is a travel guide, Fodor's 94 Travel Manager. This demo version of the software shows a traveler the city of Atlanta. It contains information about dining, lodging, services, and maps of the city (Figure 1.14).

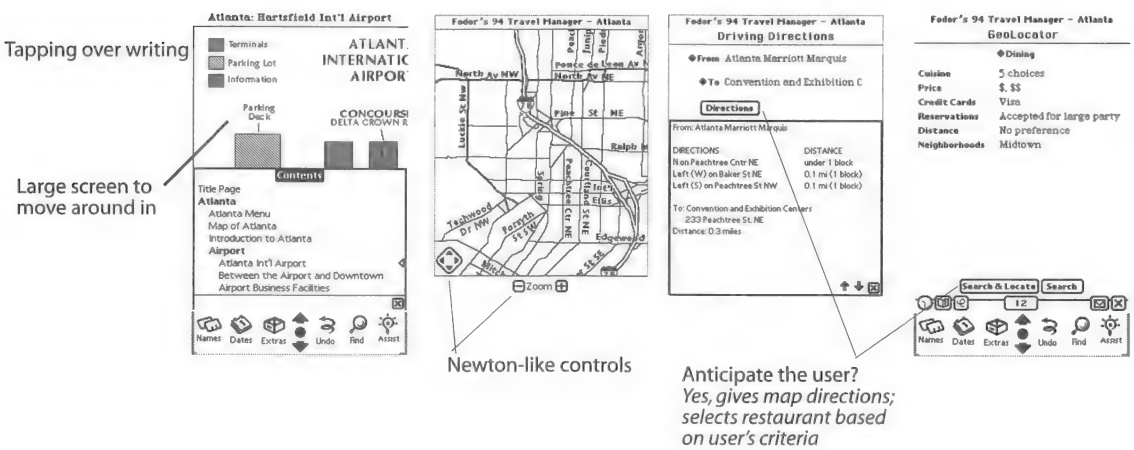


Figure 1.14 Demo version of Fodor's 94 Travel Manager.

Designing an Application—WaiterHelper

Now, it is time to discuss the application that we will be developing over the course of this book. It is called WaiterHelper™ and it performs one task—it helps waiters at the Chez Calliope Restaurant take orders. With WaiterHelper, the waiter can walk up to any table with Newton in hand and take the orders of the people sitting there.

In Figure 1.15 you can see the two main displays of WaiterHelper. On the left side is the overview; it contains a list of all the waiter's orders. Tapping on any order in the overview takes the waiter to that order. The waiter can sort the items in the overview by tapping on the different titles.

On the right side of Figure 1.15, is one table's order. The waiter taps on the chair in which the person is sitting, and takes that person's order. Tapping on another chair shows a new display. Tapping the New Item button adds a new item to that person's order.

The entire menu of Chez Calliope has been added here, so all the waiter has to do is select a category and item (for example, Soup and Gazpacho, Beverage and Iced Tea). The only place writing will occur is in the comment picker, for people who have special requests.

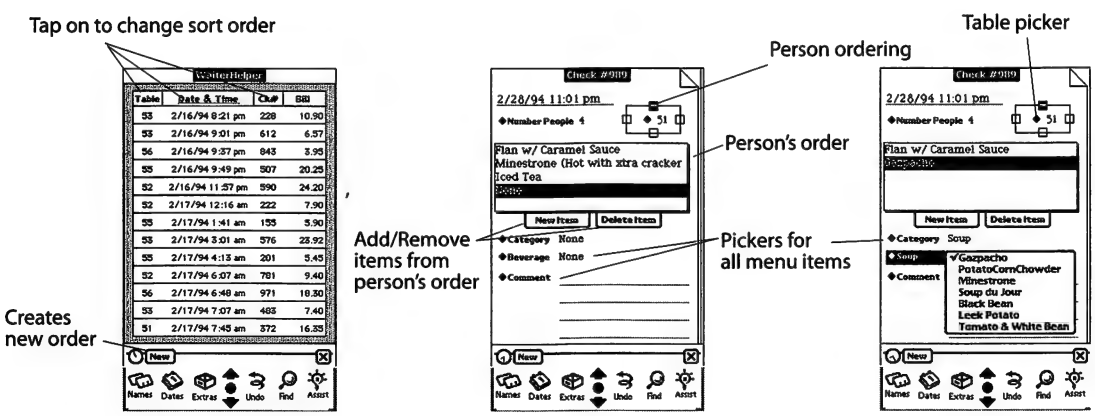


Figure 1.15 The WaiterHelper Application

Pickers are especially good to use in an application in which you know the user will be standing or walking while using the Newton. We also know that Chez Calliope is a well-lit restaurant; if it were not we would have had to make sure the text was readable in dim light.

Note: WaiterHelper is used over the course of this book to show you how to develop a Newton application. This is its primary purpose; it is a learning aid, not commercial quality software.

There are several important features that are not implemented in the final version (Routing, Undo, Find, and Intelligent Assistance) that a commercial quality application must have. The sequel to this book, *Advanced Programming for the Newton*, will cover these topics.

There are other features that have been left somewhat primitive so as to minimize already complex implementations (for example, no more than four people can be at a table).

Summary

Hopefully, you now know the three things we said you would:

- Important Newton terminology.
- The life cycle of a Newton application, from installation to its removal by the user.
- How to design a Newton application.

Given these basics, it is now time to learn more of the actual details of creating a Newton application and how each component fits into the whole process. Come on, it's Newton time.

Chapter 2

Views on the Newton

To copy others is necessary, but to copy oneself is pathetic.

—*Pablo Picasso*

Everything you see on the Newton is a *view*. Everything, be it a button, border, title, or data, is a view. Each of these views is instantiated at run time by the Newton view system using templates you create in NTK. Thus, the relationship between views and NTK templates is an important one. This chapter will introduce some of the most basic aspects of views and templates and begin explaining how views are organized. In addition, you will also learn how views are created, how templates are linked to each other, and how they are named.

Everything Is a View

As we just said, everything is a view. For instance, Figure 2.1 shows an example from the Names application where the same name card is displayed two different ways. The display on the left is principally for viewing the information. On the other hand, the elements in the name entry display on the right can be changed. In either case, each element you see is a view.

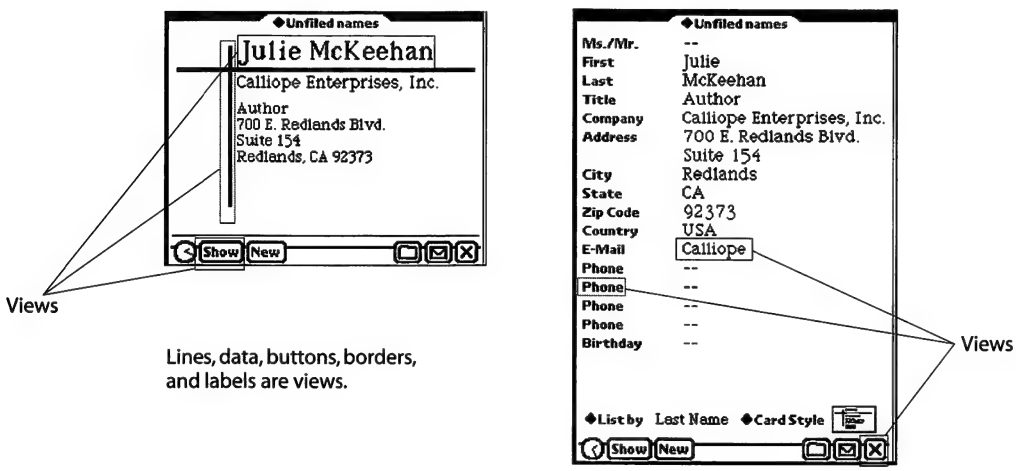


Figure 2.1 A name shown different ways.

Here is a small sampling of some of the views in the name cards:

Titles of data elements	First, Company, Zip Code
Buttons	Show, New, the close box ☒
Data elements	Julie McKeehan, 92374, USA
Adornments	Vertical and horizontal thick black lines

Views can contain anything from data to pictures. The contents of a view can remain static (for instance, an item label), change depending upon what the user writes, exist only to hold a border, or handle user taps.

Views on the Newton and Templates in NTK

Before going into greater detail on views, it is worth clarifying the relationship between the views on the Newton and what you graphically create in NTK. The views you see on the Newton are created at run time by the Newton view system. The Newton view system gets the information it needs to create these views based on templates you make in NTK. Just as the name implies, a *template* is used by the Newton view system as a blueprint for creating a view. The view system gets information about the view's size, format, and contents from the NTK template upon which it is based.

Your application is a collection of templates. When a user runs your application, views are created for those templates. Figure 2.2 shows a simple example of a Newton view and its template as created in NTK. The example shows a New button in a Newton application. This New button is based on a template created in NTK. The template has boundaries, a format, and other features associated with it. At run time, the Newton view system reads the template information and creates a view that corresponds to the New button (see the left side of Figure 2.2). The view uses proto inheritance (see Chapter 4) to inherit behavior (acting like a button) as well as data (the string "New") from its template.

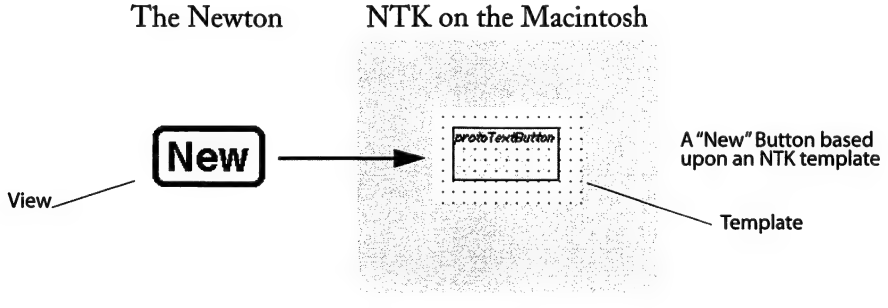


Figure 2.2 The relationship between views and templates.

Note: What is important to remember at this point is that views exist only on the Newton and that they are constructed by the view system based on information found in templates. Think of a view as a run-time instantiation of an NTK template.

A view is a frame that inherits from a template. A template is also a frame that has particular slots necessary to create a view. Newton Toolkit is designed to allow graphical editing of these types of frames. In NTK, you can modify or create templates graphically and you can alter slots within templates using NTK slot editors.

Graphic manipulation of NewtonScript data structures is one of the most important features of NTK. In standard programming environments, the data structures are traditionally modifiable only through code. You will find that graphic manipulation saves a lot of time.

Creating Templates in NTK

You create templates in NTK by drawing them out in a layout window. (For a detailed discussion on using NTK to create templates see "Creating a Layout" on page 357.) Figure 2.3 shows an NTK layout window containing a hierarchy of templates.

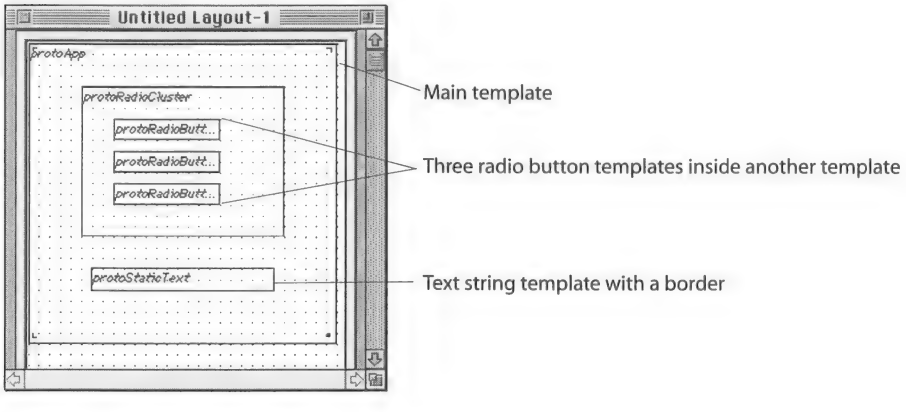


Figure 2.3 Six templates in an NTK layout window.

This example contains a view template that holds two *child* templates. The protoRadioCluster template is itself just a container template for three radio button templates. The protoStaticText template contains text surrounded by a border.

Views on the Newton

At run time, the Newton view system creates a set of views based upon these templates (see Figure 2.4).

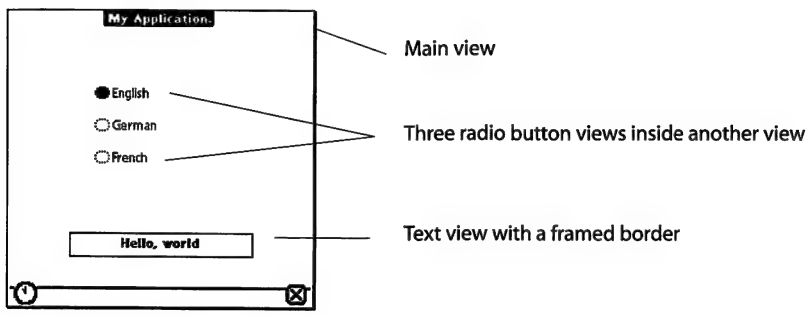


Figure 2.4 Views on the Newton at run time.

The three radio buttons are displayed above the text "Hello, world." In a fully functional program, you would expect the text string to change to "Guten tag Welt" if the German radio button were selected instead of the English. Likewise, you would expect French text if the French radio button were selected.

Although the protoRadioCluster has no visual appearance, it does have a purpose. It ensures that only one of its child radio buttons is on at a time.

View Hierarchies

As Figure 2.4 shows, views on the Newton are not completely independent units, but rather exist in some logical subgroups or hierarchies. These subgroups of views are ordered in parent–child relationships. In an application you have one main parent view (the application base view), and within that you have several levels of children. Within a particular subgroup of views, all children have the same parent. For example, the Preferences application on the Newton MessagePad has a number of child views (see Figure 2.5). There are four obvious child views of the parent view (Preferences) in the figure (the nonobvious child is the status bar containing the clock and the close box). The views labeled *Locale* and *Sound* have their own child views, and some of those children have child views as well.

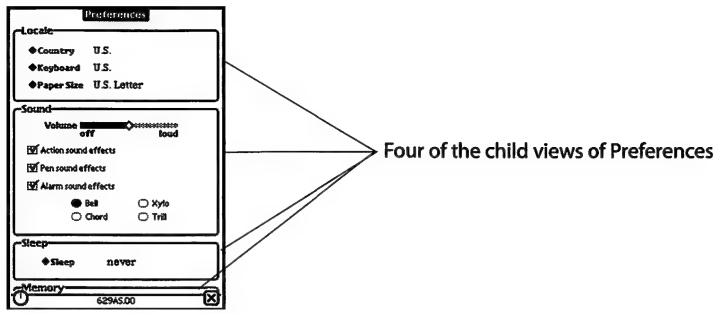


Figure 2.5 A subset of the preference settings on a Newton MessagePad.

You can get a good idea of the parent–child relationships of view hierarchies by looking at the Locale and Sound hierarchies in a tree structure (see Figure 2.6). Parent and child hierarchies can be nested several levels deep. In the figure, *Preferences* is the parent of *Locale* and *Sound*. *Locale* and *Sound* each have its own children. *Locale's* children are *Keyboard*, *Country*, and *Paper Size*. *Sound* has four children of it own. One of *Sound's* children is a cluster, *Sound radio cluster*, which in turn has four children: *Bell*, *Xylo*, *Chord*, and *Trill*. Views can have children, grandchildren, great-grandchildren, and so on, to whatever depth you desire.

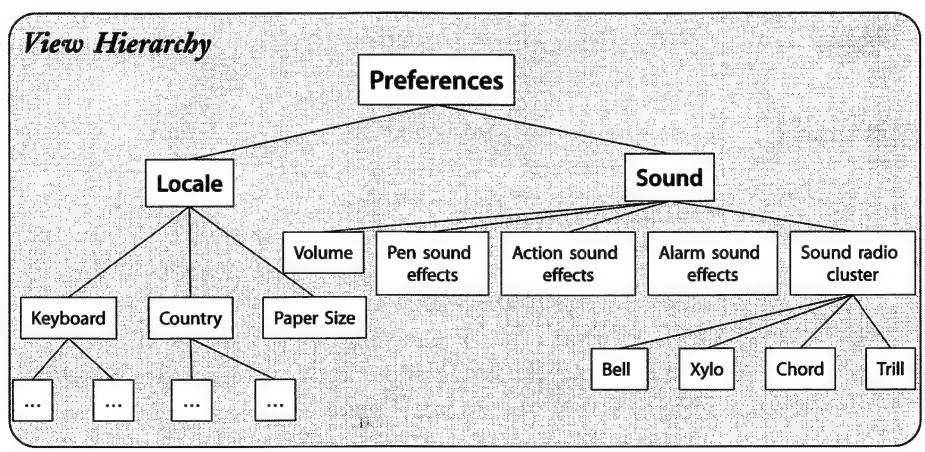


Figure 2.6 The hierarchy of some views in the Preferences application.

Rationales behind View Hierarchies

Putting views into logical subgroups allows you to manipulate whole groups of views more easily. Likewise, if you want to add some adornment to a collection of views, it is usually easiest to group them inside a parent template and then add adornment (such as a border) to the parent (for information about adorning views, see "viewFormat" on page 52). In addition to these simple uses of grouping, you can also subgroup views so that you can manage the flow of data in your application.

Grouping views improves the efficiency of the view system. Hit-testing (deciding which view has been tapped) traverses the view hierarchy. In addition, the view system keeps track of only a single view which needs redrawing. If more than one view needs redrawing, their closest common ancestor view is redrawn (along with all of its descendants—not just the specific views which needed redrawing).

Moving Views Around

Putting a number of templates in a group makes them easier to move as a unit. As an example, if you want to move a set of radio buttons around, it is much easier to do so if they are nested inside a common parent (see Figure 2.7). All you have to do is drag the parent template and the children stay in the same positions relative to one another. On the other hand, templates not inside a common parent template must be moved individually.

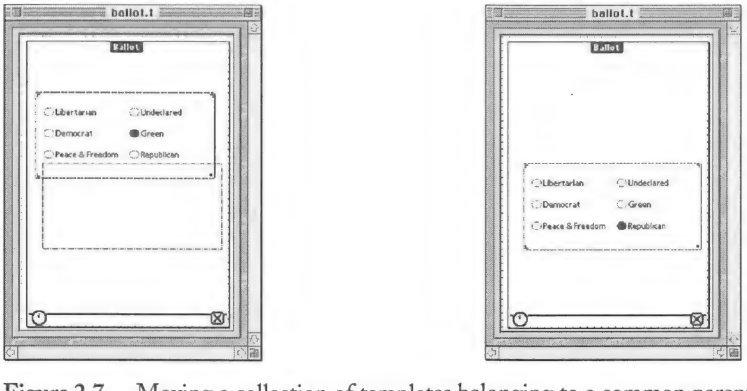


Figure 2.7 Moving a collection of templates belonging to a common parent template.

Obviously you will not put all templates into nested hierarchies. Determining what will go into subgroups of children will be based on the kind of templates you are creating and the function they have. In Figure 2.7 it makes sense to group all of the political parties into one parent since the data is closely related. In addition, the parent template, a protoRadioCluster, ensures that only one radio button is selected at once. If you added two more templates to this application, perhaps a name and an address, you would not necessarily group them together inside their own parent.

Managing Data

Another reason that you put views inside hierarchies is to manage data. The inheritance rules of NewtonScript provide views access to slots (data) within their ancestors, but not within their descendants. When you choose the view in a hierarchy that you store data in, you also affect the visibility of that data. For example, data stored in the topmost view can be accessed by any child views, their children, and so on.

View Classes

Each template is ultimately based (through inheritance) on one of a handful of primitive view classes. Each primitive view class corresponds to a C++ class object that is part of the underlying view system architecture. The view class affects the behavior and appearance of the view on the Newton. There are 10 primitive view classes:

clView	This view class serves as a very useful generic view. You will use these all the time, particularly when you need a container view or other kind of view that does not have many built-in behaviors associated with it.
clPictureView	Used whenever you want a container for a picture.
clEditView	This view can have either text or graphics children in it.

clParagraphView	A view for text that commonly has a clEditView as a parent.
clPolygonView	A view that contains shapes and commonly has a clEditView as a parent.
clKeyboardView	Used to define keyboard views that are arrays of buttons upon which you can tap.
clMonthView	Creates a month view with selectable dates.
clRemoteView	Used for a view that displays another view. Scaling is provided to enable the display of the entire remote view. You might use this when providing a page preview feature in your application.
clPickView	This view provides a picker with a pop-up list from which you can pick. Either text or graphics can be displayed.
clGaugeView	A view that provides a gauge that can be changeable or read-only.

Linking Templates

NTK allows you to use multiple layout windows to create templates that may display views in the same screen location (for a full discussion on how to create linked layouts in Newton Toolkit, see "Linking Layouts" on page 359). Linked layouts can make it easier to work with your layout windows, as layouts stay much cleaner and are easier to read.

The following example illustrates this point nicely. Imagine an application used for rating television shows. After selecting a category, an appropriate view is displayed that contains several shows to rate. As only one of the possible subviews is displayed at a time, they can occupy the same display space. If you don't use linked subviews, you need to place all the templates in the same layout window, and you end up with a cluttered mess (see Figure 2.8).

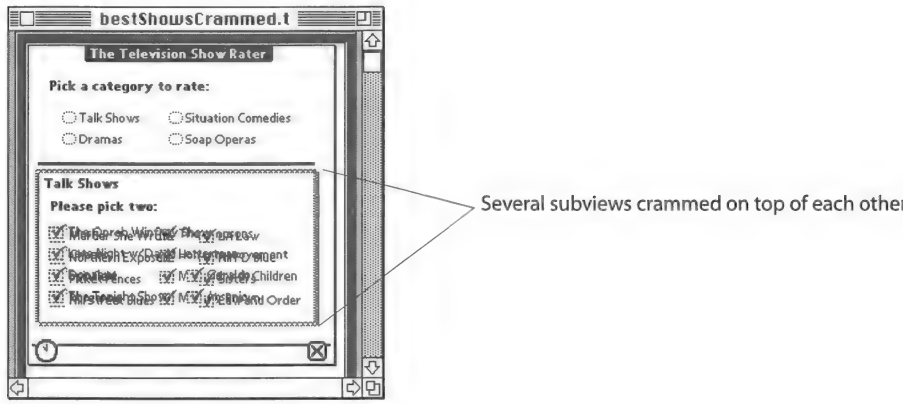


Figure 2.8 A cluttered layout window.

By using linked subviews, you can place each of the subviews in a different layout file. The end result is clean, well-ordered layouts. Figure 2.9 shows the same television show rating program using linked subviews.

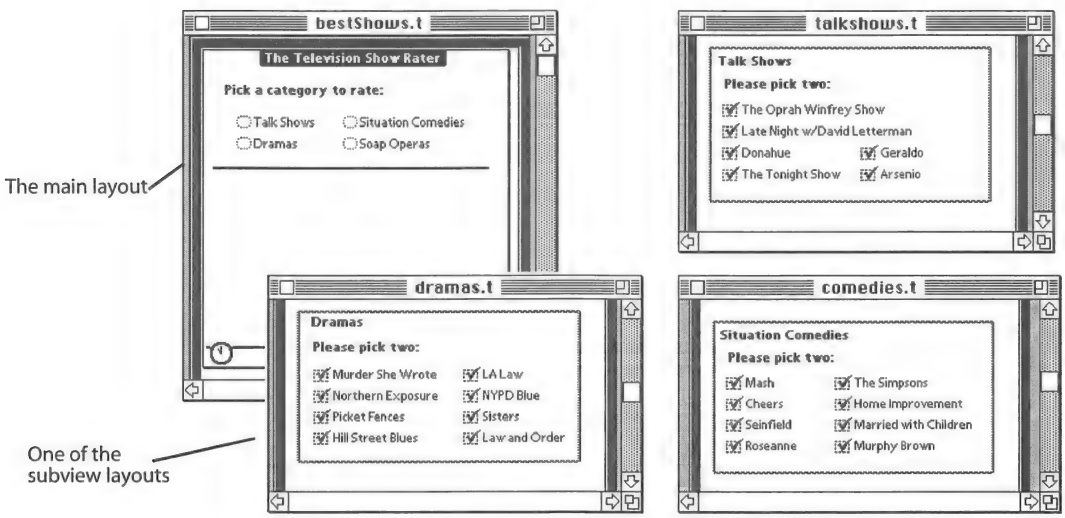


Figure 2.9 Well-ordered layout windows.

When NTK builds your project, it reads all the linked layouts and puts them together in one large hierarchy of templates. That is, after NTK has built your application, there are no linked layouts. Instead, they are used only within NTK as a way to prevent clutter. Linked layouts are a figment of NTK's imagination. They do not actually exist on the Newton.

Linked templates are not a mechanism for reuse. A layout file can be linked only once. Thus, linked layouts don't provide a way to use the same layout in more than one place in a project, or across projects. In order to reuse templates, you use protos. For more information on protos, see Chapter 4.

Naming Templates

In NTK, you can attach names to the templates that you create (see "Naming Templates" on page 365 for detailed instructions on how to name templates in NTK). There are several reasons for naming templates. The simplest reason is it makes it easier to tell what you are editing. For instance, if you are working with eight different checkboxes, it is easier to distinguish them if you name them. Figure 2.10 shows template browsers with and without naming.

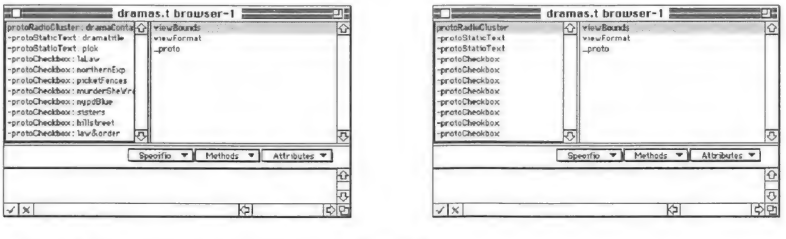


Figure 2.10 Named versus unnamed templates.

Sending Messages to Views

Another important reason for naming templates is the ability to send messages to specific views (note that this also requires declaring the views; see "Declaring Views" on page 180). For instance, in the television show rating application you would need to open and close the category subviews depending upon which category radio button the user taps. To do this, you would send `Open` and `Close` messages using the name of the views. This topic is discussed in more detail in "Open()" on page 179.

What to Name Templates

In general, you should name your templates beginning with a lowercase letter, and then include only letters and numbers in its name. Be aware that NewtonScript is case-insensitive, so don't name one template "Label" and another template "label" and expect NewtonScript to distinguish them from one another.

Creating the Views of WaiterHelper

It is now time to start working on the sample application, WaiterHelper. Throughout the book, we will be building the application in stages.

Creating the Project

To begin, create a project named WaiterHelper (see "NTK's Window menu." on page 354). Create a layout window (see "Creating a Layout" on page 357), name it "MainLayout.t", and add it to the project (see "Adding Files to a Project" on page 356).

Creating the Main Layout

Within "MainLayout.t", draw a protoApp roughly the size of the Newton screen. Create a browser window and edit the title slot of the protoApp to contain "WaiterHelper" (see "An Introduction to the Browser" on page 367). Figure 2.11 shows the layout file at this stage of development.

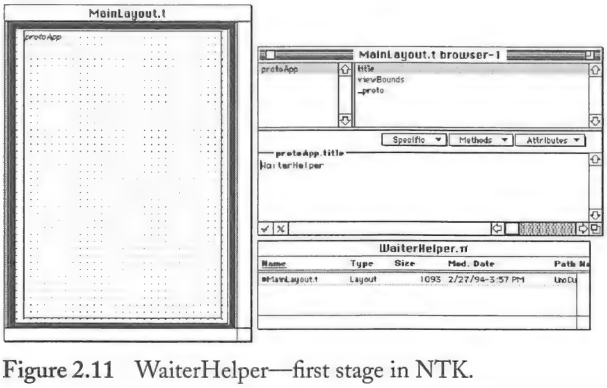


Figure 2.11 WaiterHelper—first stage in NTK.

Before adding anything else, build the application and try it on the Newton. We recommend that you iterate often during application development—don't do too much work without trying it. Also, the NTK build and download process is very fast and therefore conducive to this approach. With that advice in mind, build your application (see "Building a Package" on page 380) and then download it ("Downloading a Package" on page 381).

Figure 2.12 shows what your application should look like on the Newton after you have downloaded it.

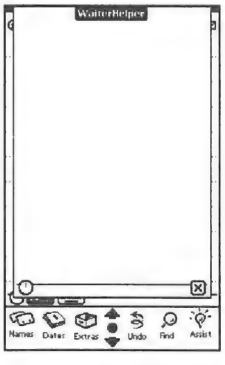


Figure 2.12 WaiterHelper—first stage on Newton.

The final WaiterHelper application will have two distinct views: the overview and the detail view. Since only one view will be displayed at a time, but in the same screen space, it makes sense to use linked layouts. Thus, WaiterHelper will have three layout files: one main file and two subview layout files. The subview layouts are linked to the main one. One subview layout is for the overview template, and the other is for the detail template.

Until the next chapter, however, we have no way to make only one display at a time. Thus, we'll concentrate on the detail template in this chapter, and add the overview template in the next chapter.

Creating the Detail Template

Create a layout window, name it "Detail.t", and add it to the project. The detail layout is composed of a container clView with a number of different children, which include pictures, a view displaying an order, and various pickers. Figure 2.13 shows the complete layout with all the templates as well as the template names we'll use.

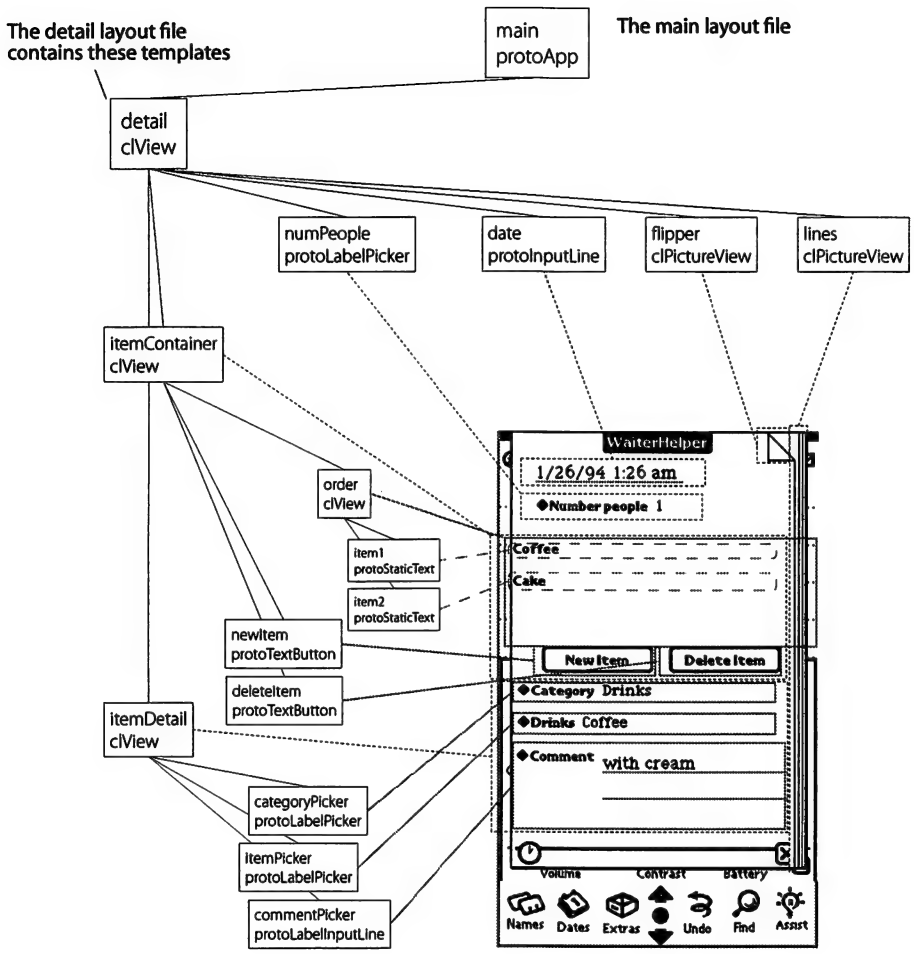


Figure 2.13 The template hierarchy of the detail layout.

To begin with, create three templates and set these various slots:

1. Create the topmost container template, detail in the "Detail.t" layout. Draw a clView as wide as the protoApp, but not quite as tall (to reserve room for the status bar at the bottom). Name this template "detail" and then create a browser window for it.

2. Draw a protoInputLine template in the top left and name it "date". Edit the text slot of date to contain the string, `1/26/94 1:26 am.`

3. Draw a protoLabelPicker below the date template and name it "numPeople".

4. Change the text slot to `Number people`.

5. Change the labelCommands slot to `["1", "2", "3", "4"]`. The picker should now display the strings 1, 2, 3, 4.

Figure 2.14 shows the detail template so far.

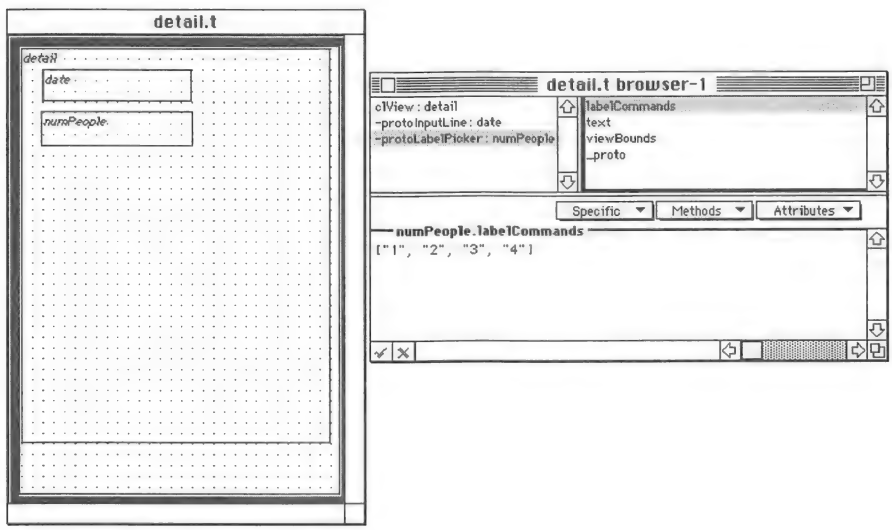


Figure 2.14 Detail template after adding date and numPeople.

Linking the Detail Layout

Now is a good time to test the application. Before building the application, however, you need to link the detail layout to the main layout.

1. To do so, draw a LinkedSubView in the main layout (the size and location don't matter).

2. Select the linked layout and link it to "detail.t" (see "Linking Layouts" on page 359).

Figure 2.15 shows what the main layout should look like after linking.

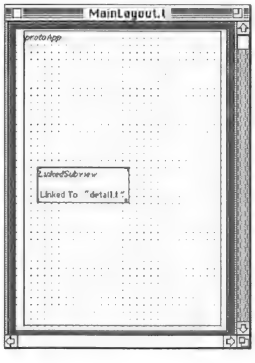


Figure 2.15 MainLayout.t after linking to "overview.t".

It is now time to build, download, and run the application (see Figure 2.16). You should be able to scrub or write in either the date or numPeople view. In addition, you can select an item from the numPeople picker. You may need to adjust the sizes and locations of your templates if they are too large or too small.

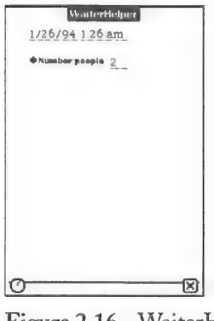


Figure 2.16 WaiterHelper detail (with detail and numPeople) on the Newton.

Adding the OrderContainer

Now let us create the orderContainer template, the part of the detail template that displays one person's complete order. The orderContainer also has two buttons templates: one to create a new item and one to delete an item (see Figure 2.17).

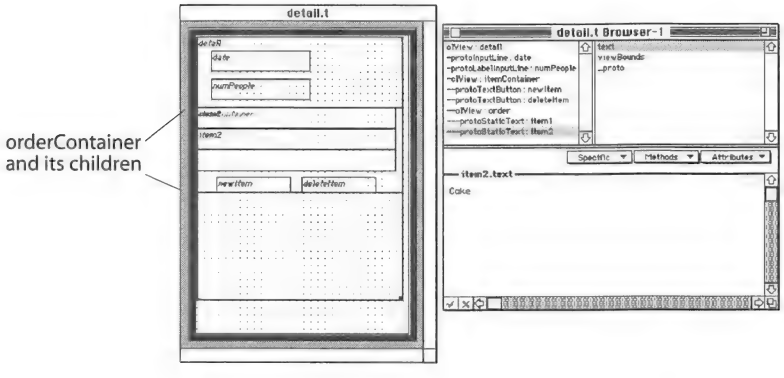


Figure 2.17 WaiterHelper detail template after adding orderContainer.

Now that you know what you are making, follow these steps:

1. Draw a clView below the numPeople template and make it the same width as the detail template. Make it high enough to hold all of its children templates (about one third the height of the detail template). Name this clView "orderContainer".

2. Within orderContainer, draw two protoTextButtons. Place them side by side near the bottom of the orderContainer. Name one "newItem" and the other "deleteItem". Set the text slot of the first to `New Item`, and set the second to `Delete Item`.

3. Draw a clView inside orderContainer starting at the top left corner and ending just above the two buttons. Inset it slightly from order-Container on the right. Name this template "order". This template will contain item1 and item2.

In the next chapter, we'll add a shadowed border to this template (thus the inset on the right), but for now it has no visual appearance.

4. Draw two protoStaticTexts within order, each of which should extend the full width. One should start at the top, the other should be just below it. Name them "item1" and "item2", respectively. Set the text slot of the first to `Coffee`, and set the second to `Cake`.

The time has come to test the application on the Newton again. After you build, download, and run, the application should be similar to the one in Figure 2.18. The buttons should flash when you tap them, but not much else will happen.

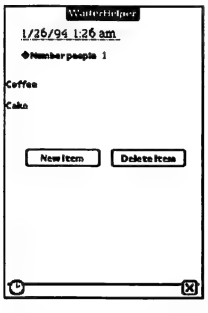


Figure 2.18 WaiterHelper detail (with orderContainer) on the Newton.

Adding ItemDetail

Next, we'll add itemDetail. This template contains three pickers: one for the menu category, one for the item within a category, and one for a comment about the item. Figure 2.19 contains the detail template as it will look after you have added itemDetail and its three child picker templates.Here are the steps to add these new templates:

1. Draw a clView right below the order template and all the way to the bottom of the detail template. Start the template at the left of the detail template, but don't extend it all the way to the right side (you'll need space for lines). Name the template "itemDetail".

2. Draw two protoLabelPickers and one protoLabelInputLine. Name them and set the label, text, and labelCommands as in Table 2.1.

	1st protoLabelPicker	*2nd protoLabelPicker*	*3rd protoLabelInputLine*
name	"categoryPicker"	"itemPicker"	"commentPicker"
label			`Comment`
text	`Category`	`Item`	`with cream`
labelCommands	`["Drinks", "Desserts", "Entrees"]`	`["Coffee", "Tea", "Cola"]`	`["Hot", "Cold"]`

Table 2.1 Settings for the protoLabelPickers and protoLabelInputLine in itemDetail.

The time has come to test on the Newton again. Build, download, and run the application (see Figure 2.20). The buttons should flash when you tap them although, of course, they don't do anything yet.

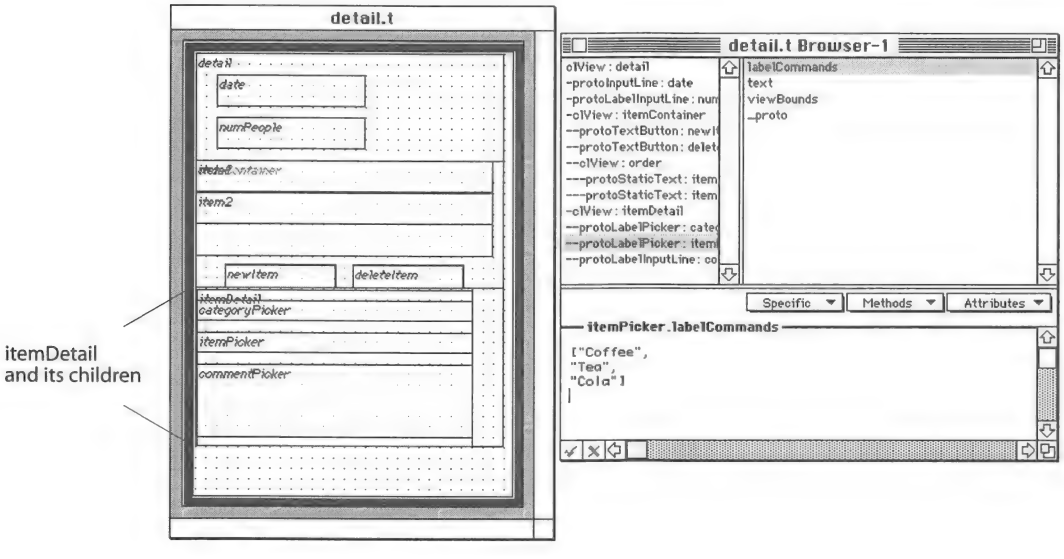


Figure 2.19 WaiterHelper detail template after adding itemDetail.

Figure 2.20 WaiterHelper detail (with itemDetail) on the Newton.

Adding the Pictures

Now we need to add the turned-page picture (the flipper) at the top right, as well as the lines down the right side. Rather than having you create the pictures, we've provided them on the disk that comes with this book. The files are called "flipSlip" and "noteLines" and are located in the Pictures folder on your disk. Copy them to the folder containing your WaiterHelper project. Next, add the two picture files to your project. Figure 2.21 shows the detail layout after adding these two pictures (and after nudging them into their proper places).

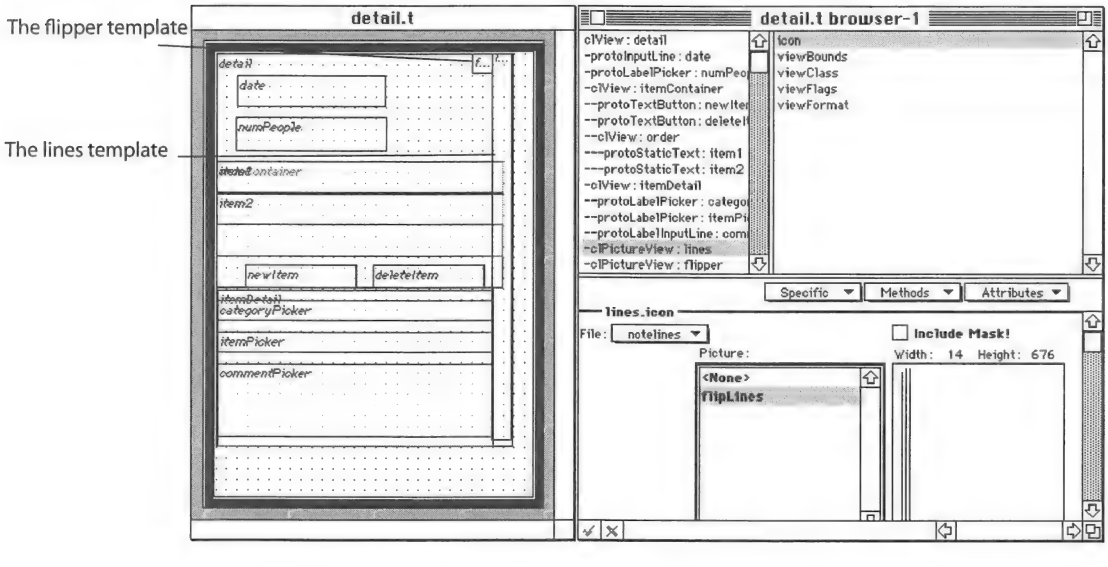


Figure 2.21 WaiterHelper detail template after adding pictures.

Before creating your new templates, make sure that the width of the detail template is the same as the width of the protoApp template in the main layout. If the detail template is wider, you'll lose anything that's on the right edge (in particular, the lines).

Adding Lines

Here are the steps for drawing the line templates:

1. Draw a clPictureView on the right side of the detail view. Make it extend all the way from top to bottom. Name it "lines".
2. Now you need to specify the picture for "lines". Edit the icon slot of the lines template first. Select "notelines" from the File popup menu, and then choose "fliplines" from the Picture list (see Figure 2.22). Notice that the file name and picture name need not be identical; in this case they are different.

Adding the Note Corner

1. Draw another clPictureView at the top of the detail template, just to the left of "lines". It should be a squarish shape and named "flipper".
2. Now specify the picture for the flipper template, edit the icon slot, and select "flipSlip" from the File popup menu and "flipslip" from the Picture list (see Figure 2.23).

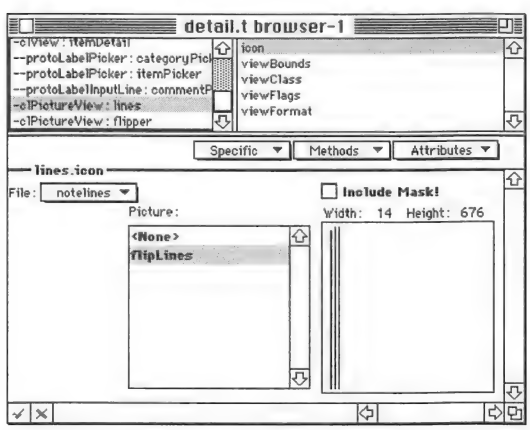


Figure 2.22 Choosing the picture for the lines template.

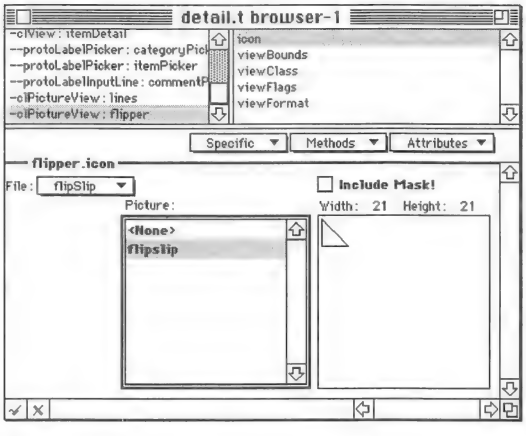


Figure 2.23 Choosing the picture for the flipper template.

Now that you've created the last two templates, build, download, and run the application. You'll probably find that the pictures are not in the exact location you want them. You can nudge a template one pixel at a time by selecting it and then pressing one of the arrow keys to fix the problem.

Note: The picture files you add to your project are files containing named 'PICT' resources. To create your own, do the following:

Use your favorite drawing program to create a picture.
Copy it to the clipboard.
Launch ResEdit (a resource-editing program from Apple).
Create a new file from within ResEdit.
Create a new 'PICT' resource from within ResEdit.
Paste in the picture.
Use Get Resource Info to give the resource a name.

A file can contain more than one 'PICT' resource; just make sure they have unique names.

Figure 2.24 shows the running application. Notice that the lines draw on top of the close box. This is something you will fix in the next chapter.

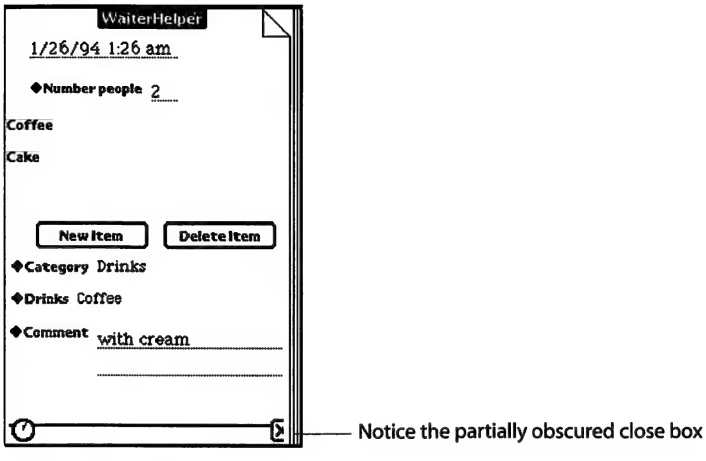


Figure 2.24 WaiterHelper detail (with pictures) on the Newton.

Summary

In this chapter you got an introduction to views on the Newton and their relationship to templates in NTK. You also learned that templates are structured into parent–child hierarchies. We also discussed linking layouts so that you can have cleaner templates and we talked about how and why to name templates. Naming templates allows you to distinguish more easily between like items and allows you to send messages.

At the end of the chapter we started creating an application that we will be building throughout this book. You saw how to draw out templates and place them in logical parent–child hierarchies. We even added some adornments to the application in the form of a folded-page corner for turning pages and vertical lines to add a feeling of depth.

Chapter 3

Skeleton of a View

Eat to please thyself, but dress to please others.

—*Benjamin Franklin*

Vestus viewum reddit.

—*unknown*

You can go a long way toward the final look of your views without writing any code. All you have to do is edit various template slots from within NTK. In these slots, you can specify what type of frame, fill, and shadow a view has. You can also set the type of font the view will use for displays, and you can set the size and location of the view.

NTK and the Newton view system also provide you with some powerful view justification tools. You can specify the placement of one view relative to another in a myriad of ways. While understanding the relationship between the view coordi-

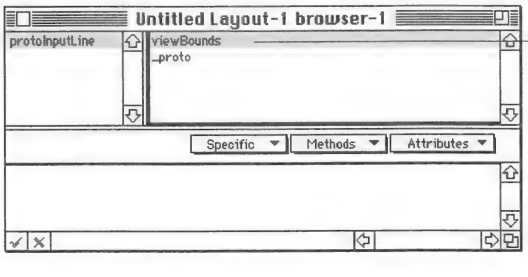
nates (found in the `viewBounds` slot) and the justification (found in the `viewJustify` slot) can take some getting used to, it is well worth the effort. After detailing the justification features of NTK and the view system, this chapter also describes why and how you would use justification in your application design.

Once you have set all the view features found in various common slots and set the size and justification, you will have created all the basic components of the view. Its skeleton will be in place and ready for your code sinew. Lastly, we will implement these various view settings in the WaiterHelper application.

What's in a View

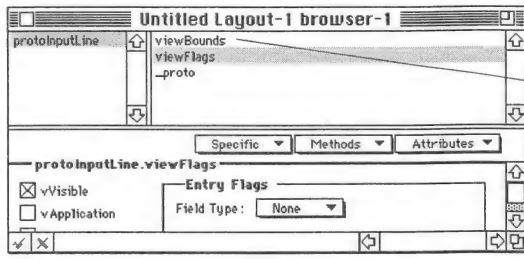
A view, as we said in Chapter 2, is based upon a template that you create in NTK. This template is just a NewtonScript frame that you can modify using NTK's graphical template editor. Among the slots that this template contains are slots that govern the standard appearance and behaviors of a view.

Four of these slots are a part of all templates. These are `viewBounds`, `viewClass`, `viewFlags`, and `viewFormat`. In certain cases, however, these slots do not appear in the slot editor (see Figure 3.1). For instance, when you create a template based on a system proto, the system proto may already have those slots filled with predefined values. Thus, when you create a template, you inherit those values. In fact, if you add a slot to the template (like `viewFlags`), you are overriding the template's inherited slot values (see Figure 3.2).



For a `protoInputLine` template, only `viewBounds` is created within the template. The other three common slots `viewFlags`, `viewClass`, and `viewBounds`, are inherited.

Figure 3.1 A template with inherited common slots.



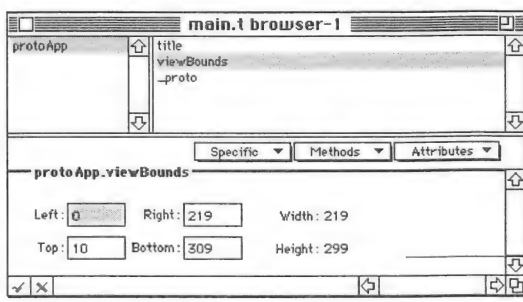
A `protoInputLine` template with a new slot, `viewFlags`, added to it. The new `viewFlags` slot overrides the slot inherited from the proto.

Figure 3.2 A template with a new slot that overrides the original.

Common View Slots

Again, the slots that are common to all templates are `viewBounds`, `viewFlags`, `viewFormat`, and `viewClass`. Let's look at each in turn.

viewBounds



The rectangle slot editor of NTK

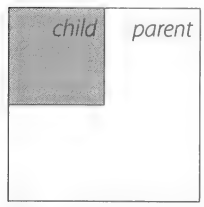
Figure 3.3 The `viewBounds` slot of a `protoApp`.

The `viewBounds` slot defines the size and coordinates of a view (see Figure 3.3). This slot is itself a frame that contains four slots, one slot each for the left, right, top, and bottom coordinates of a view. Later in this chapter you will learn how the values in `viewBounds` depend upon the values found in the `viewJustify` slot. For now, simply realize that these `viewBounds` values are integer values that specify view location and size *relative to some other view*. When no other view is specified, the view's parent is used by default.

Thus, if you have parent and child views with these `viewBounds`:

```
    Parent:                Child:
left:     0,             left:     0,
right:   80,             right:   40,
top:      0,             top:      0,
bottom:  80              bottom:  40
```

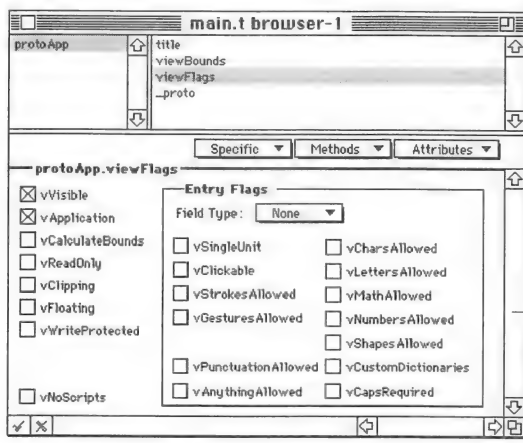
you will have a child view half as wide and high as its parent and nestled in the top left corner (see Figure 3.4).



parent `viewBounds`: `left: 0, right: 80, top: 0, bottom: 80`
child `viewBounds`: `left: 0, right: 40, top: 0, bottom: 40`

Figure 3.4 A child view inside a parent view.

viewFlags



The `viewflags` slot editor of NTK

Figure 3.5 The `viewflags` slot of a protoApp.

The `viewFlags` slot contains other settings that govern the behavior of the view. Each of these settings is a constant defined by a bit flag (see Figure 3.5). We will not cover every setting, but rather focus on the most important ones.

Flags that Affect the Behavior of the View

`vVisible`	When this is set, the view is visible when the application is run. If it is not set, the view is invisible and can only be opened by sending the view an `Open` or `Show` message (note that the view will need to be declared to its parent before it can receive any messages).
`vApplication`	When this is set, the view is flagged as able to respond to taps on the scroll up and down arrows and the overview button. If you create a template based upon a `protoApp`, this flag is set automatically.
`vClipping`	If this is set, everything drawn in this view or a child view is clipped to the bounds of this view. This option defaults to off to speed drawing on the Newton. Unless you specifically need a view clipped, you should not set this.
`vClickable`	This must be set for a view to explicitly handle user taps. For instance, you will use this for a row displayed in an overview of an application where tapping takes the user to a different close-up view.

Flags that Affect Input

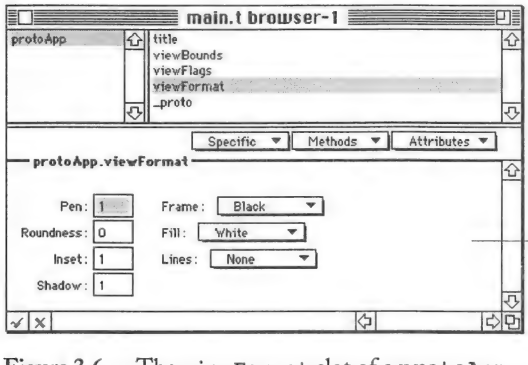
`vGesturesAllowed`	Gestures like scrub, tap, and double-tap are understood by the view. If a view accepts input it would usually accept gestures.
Phone Field	This field ensures that phone numbers are recognized. A special phonepad keyboard is provided for editing.
Date Field	This ensures that dates are recognized. A special date keyboard is provided for editing.
Name Field	Uses the Names dictionary from the built-in Names application as the primary dictionary for word recognition.

Flags that Affect Text Input

`vCharsAllowed`	Dictionaries are used in recognizing words.
`vLettersAllowed`	Words are recognized character-by-character without using dictionaries. License plate numbers are a good example of the type of data for which you would set this flag.
`vMathAllowed`	Uses recognition for mathematical symbols. If you set this flag it is best to limit other types of recognition.
`vCustomDictionaries`	Ensures that custom dictionaries are used with character recognition. You must include a `dictionary` slot in the template that refers to the custom dictionary. You can also use this flag to restrict the use of built-in dictionaries. Examples of uses for custom dictionaries would include large numbers of specialized words such as medical terminology.

viewFormat

The `viewFormat` slot defines the appearance of the view, including what fill color, border type, and inset it uses (see Figure 3.6). Let's look at each view aspect that you can set with `viewFormat`.



The `viewFormat` slot editor of NTK.

Figure 3.6 The `viewFormat` slot of a `protoApp`.

The View Frame

You can select a frame of any shade from white to black or have no frame at all. The thickness of the frame is governed by the Pen setting. A value of 1 equals a frame one screen pixel thick. Setting Roundness gives you rounded corners on the frame. Using Inset places the frame on the inside of the view (framing on the outside is the default). Adding shadow to a view creates the visual effect of depth.

The View Fill

If you will be dirtying a view often, don't set its fill to none. A none fill is slower in this case since the view system has to redraw views that are behind the transparent view. On the other hand, if a view is redrawn because its parent view is dirty, then doing an unnecessary fill can be slower.

The View Lines

You specify what type of lines you want to display in a view. These can be anything from white to black or you can use a custom pattern.

The View Hilite

This specifies what type of highlighting is used when a view is tapped upon.

Some viewFormat Examples

Figure 3.7 contains several different view formats that have been applied to a 60x30 empty view along with an NTK preview of each.

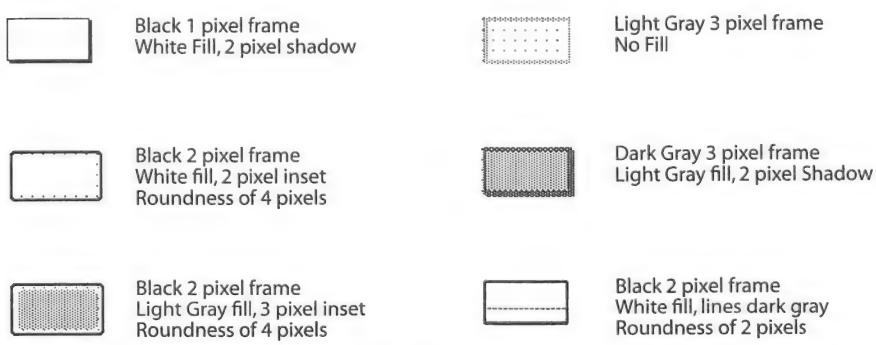


Figure 3.7 Different types of `viewFormat`.

viewClass

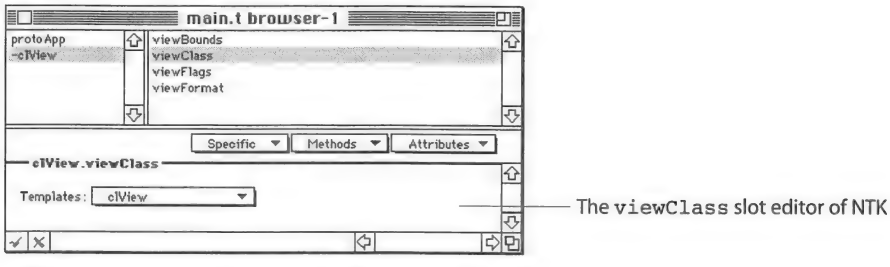


Figure 3.8 The `viewClass` slot of a `clView`.

This slot contains the primitive view class that underlies the view (see Figure 3.8). For more information on the types of view classes, see "View Classes" on page 30.

Other View Settings

There are some other modifications that you can make to a view. You can create some animation effects using the `viewEffect` slot, and you can set the display font, among other things. Let's look at each of the remaining view settings.

viewEffect

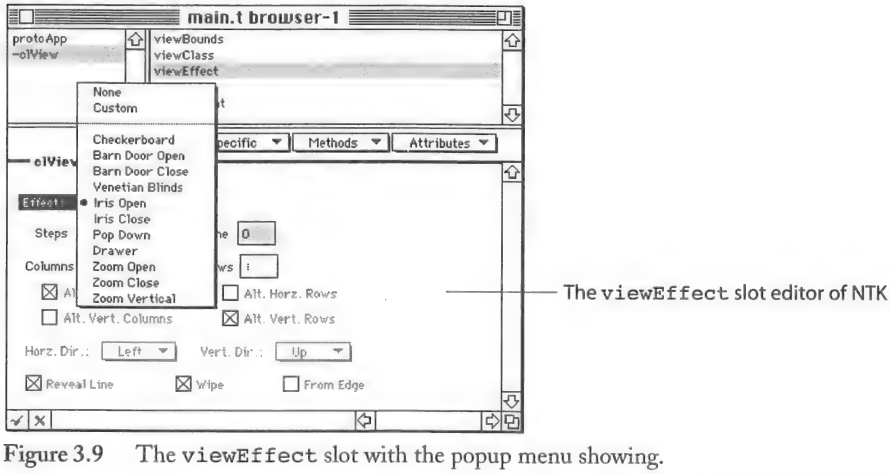


Figure 3.9 The `viewEffect` slot with the popup menu showing.

A `viewEffect` slot (see Figure 3.9) in a template causes a visual effect when the view associated with the template is opened or closed. These visual effects are most similar to the transitions (or wipes) used in some movies when changing from one scene to another. For example, you can have a view that opens like the iris of a camera. When that view is closed, it uses the visual effect in reverse, like the closing of the iris of a camera. Here is the list of predefined visual effects:

- `fxCheckerboardEffect`
- `fxBarnDoorOpenEffect/fxBarnDoorCloseEffect`
- `fxVenetianBlindEffect`
- `fxIrisOpenEffect/fxIrisCloseEfffect`
- `fxPopDownEffect`
- `fxDrawerEffect`
- `fxZoomOpenEffect/fxZoomCloseEffect`
- `fxZoomVerticalEffect`

You might want to use `viewEffect` as a slot within your application base view, so that your application will have a particular visual effect whenever it is opened or closed. You can also add `viewEffect` slots to other views that you open and close.

Note: Other effects are possible from within NewtonScript. In particular, `SlideEffect` is commonly used for scrolling.

---

viewFont

You can specify the view's font in the `viewFont` slot. Several fonts are built into the system ROM. You can use them by specifying either a font frame or an integer specifying a font.

Table 3.1 contains a listing of font frames available in ROM.

Font Frame Constant	*Font Frame*
Font Family: Espy	
ROM_fontsystem9	{family: ROM_espyfont, face: kFaceNormal, size: 9}
ROM_fontsystem9bold	{family: ROM_espyfont, face: kFaceBold, size: 9}
ROM_fontsystem9underline	{family: ROM_espyfont, face: kFaceUnderline, size: 9}
ROM_fontsystem10	{family: ROM_espyfont, face: kFaceNormal, size: 10}
ROM_fontsystem10bold	{family: ROM_espyfont, face: kFaceBold, size: 10}
ROM_fontsystem10underline	{family: ROM_espyfont, face: kFaceUnderline, size: 10}
ROM_fontsystem12	{family: ROM_espyfont, face: kFaceNormal, size: 12}
ROM_fontsystem12bold	{family: ROM_espyfont, face: kFaceBold, size: 12}
ROM_fontsystem12underline	{family: ROM_espyfont, face: kFaceUnderline, size: 12}
ROM_fontsystem14	{family: ROM_espyfont, face: kFaceNormal, size: 14}
ROM_fontsystem14bold	{family: ROM_espyfont, face: kFaceBold, size: 14}
ROM_fontsystem14underline	{family: ROM_espyfont, face: kFaceUnderline, size: 14}
ROM_fontsystem18	{family: ROM_espyfont, face: kFaceNormal, size: 18}
ROM_fontsystem18bold	{family: ROM_espyfont, face: kFaceBold, size: 18}
ROM_fontsystem18underline	{family: ROM_espyfont, face: kFaceUnderline, size: 18}
Font Family: Geneva	
simpleFont9	{family: ROM_genevafont, face: kFaceNormal, size: 9}
simpleFont 10	{family: ROM_genevafont, face: kFaceNormal, size: 10}
simpleFont 12	{family: ROM_genevafont, face: kFaceNormal, size: 12}
simpleFont 18	{family: ROM_genevafont, face: kFaceNormal, size: 18}
Font Family: New York	
fancyFont9	{family: ROM_newyorkfont, face: kFaceNormal, size: 18}
fancyFont 10	{family: ROM_newyorkfont, face: kFaceNormal, size: 18}
fancyFont 12	{family: ROM_newyorkfont, face: kFaceNormal, size: 18}
fancyFont 18	{family: ROM_newyorkfont, face: kFaceNormal, size: 18}

Table 3.1 Fonts available on the Newton MessagePad.

When you specify a font with an integer, some of the bits specify the family, some the face, and some the size. You usually use one of the following constants for the family and size:: `userFont9`, `userFont10`, `userFont12`, `userFont18`, `simpleFont9`, `simpleFont10`, `simpleFont12`, `simpleFont18`, `fancyFont9`, `fancyFont10`, `fancyFont12`, `fancyFont18`. Then, you add a constant for the style: `tsPlain` (whose value is 0), `tsBold`, `tsItalic`, `tsUnderline`, `tsOutline`, `tsSuperScript`, `tsSubScript`

Miscellaneous Settings

There are a number of other slots you can add to a template to affect how the view is displayed. Here is a brief description of each one:

`viewOriginX`	Sets the amount of horizontal offset used when displaying the contents of a view. When set to 0, the contents of a view are displayed starting at the very left.
`viewOriginY`	Sets the amount of vertical offset used when displaying the contents of a view.
`viewFillPattern`	Sets a custom fill pattern for a view. The fill pattern is a 64-bit (8x8) binary object.
`viewFramePattern`	Sets a custom frame pattern for a view. The fill pattern is a 64-bit (8x8) binary object.
`viewTransferMode`	Specifies the transfer mode that is used for drawing in the view. The possible choices are modeCopy, modeOr, modeXor, modeBic, modeNotCopy, modeNotOr, modeNotXor, modeNotCopy, and modeMask.
`copyProtection`	Offers template copy protection.
`declareSelf`	The value of this slot is a symbol (commonly the symbol `'base`). This symbol will be created as a slot in the view of this template at run time and will point at the view itself.

Why Use Justification?

Here are some reasons for using the justification tools that NTK provides to you.

Newton Is a Family of Products

Each Newton family member runs the Newton operating system. The particular hardware characteristics are not guaranteed—not size, weight, number of PCMCIA slots, and certainly not screen size. So, if you want applications you create for the Newton to display correctly, don't assume a fixed screen size. Using the view justification tools provided, you can write your applications so that they will display correctly even in the face of such unknowns.

Note: The size of the Newton MessagePad and MessagePad 100 is 240x336 pixels, an unusual choice. The MessagePad 110 has a screen size of 240x320 pixels, exactly one-quarter of the common 640x480 screen size. Using a common screen size reduces costs.

By using justification correctly, you can ensure that your views will display differently based on the screen size of the Newton. As an example, consider the detail view of our WaiterHelper application. Imagine that it is run on a slightly taller screen. What would you like to see happen? We think that the comment view should stretch in the case of a taller screen, and that the other views should stay the same size. The views in our application are customized to do just this.

Design Changes Are Easier

The second reason to use view justification is that design is easier. Application view design is best done as an iterative process. You make an initial design that will most likely require rearrangements of the sizes and locations of things. By using the justification system, you can adjust one view and have a number of other views rearrange themselves to the correct size as well. For example, if you have six similar checkboxes that need lengthening, you should be able to adjust one and have the other five templates resize to reflect these changes (see Figure 3.10) if you have set justification for each of the checkboxes correctly.

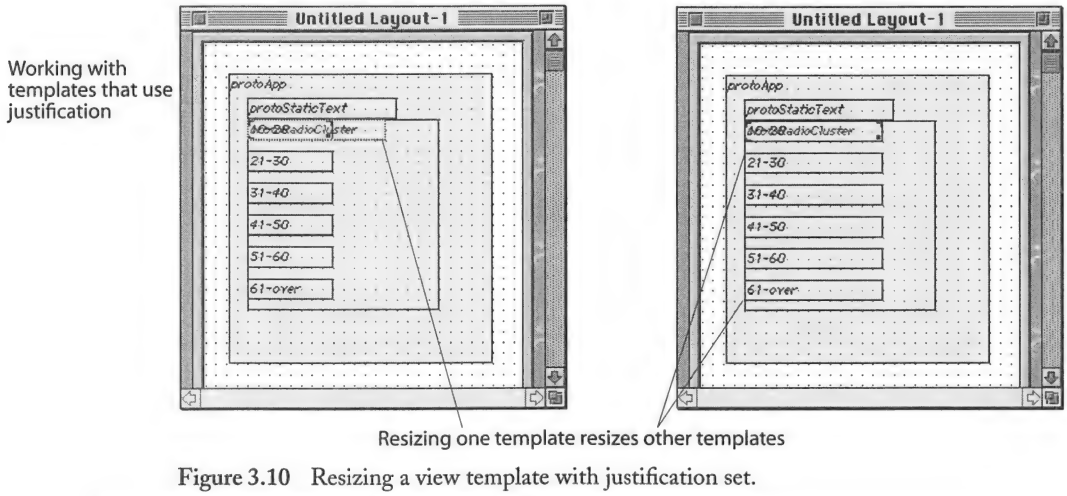


Figure 3.10 Resizing a view template with justification set.

Using Justification

There are several aspects to view justification. You need to determine whether you are going to justify relative to a parent or a sibling and you have to decide on which side(s) of that view you are going to base the justification (top, left, bottom, or right). Having set the justification to your liking you then need to modify the `viewBounds` slot so that the size and location are correct.

How Justification Is Done

All justification is done using the `viewJustify` slot in a particular template (see Figure 3.11). The `viewJustify` slot contains an integer that is used as a bit-field—some bits for horizontal justification and some for vertical justification.

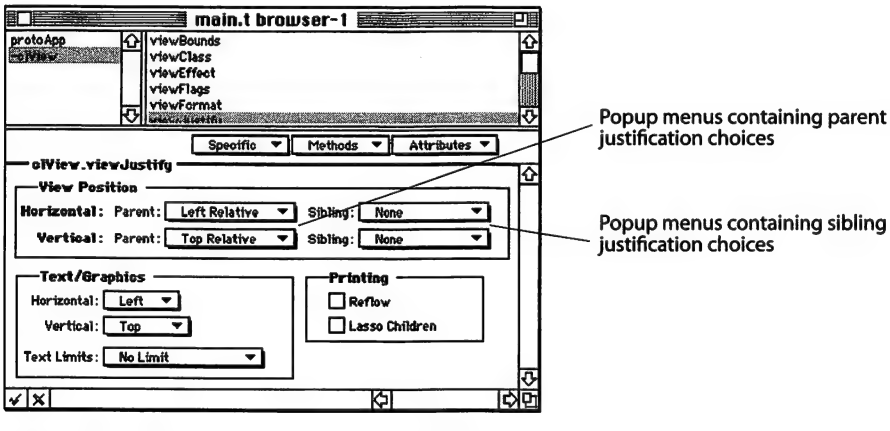


Figure 3.11 The `viewJustify` slot of a clView.

Ways to Justify a View

You can justify a view relative to another view in two different ways:

- with respect to its parent
- with respect to its sibling view

Justifying to either of these, you can specify relative to the left, right, center, top, bottom, full width, or full height of that view.

Using Parent Justification

When you justify a view relative to a parent, the view's `viewBounds` are relative to the parent's `viewBounds` coordinates. You can justify a view horizontally in any of these ways:

- `parentRelativeLeft/parentRelativeRight`
- `parentRelativeCenter`
- `parentRelativeFull`

You can also justify a view vertically using similar settings:

- `parentRelativeTop/parentRelativeBottom`
- `parentRelativeCenter`
- `parentRelativeFull`

As you can see, any boundary of a view can be used as the basis of the justification. Which justification you choose depends on exactly what effects you want. In some cases, you will also find that more than one choice will work equally well.

It is important to understand that once you add a `viewJustify` slot to a view template, its `viewBounds` slot's values are interpreted relative to the view justification you specify. In fact, you can't even interpret the meaning of the `viewBounds` until you know the justification of a view. In Figure 3.12 you can see a view, *child*, whose `viewBounds` never changes even though its position does. What changes is the type of justification specified.

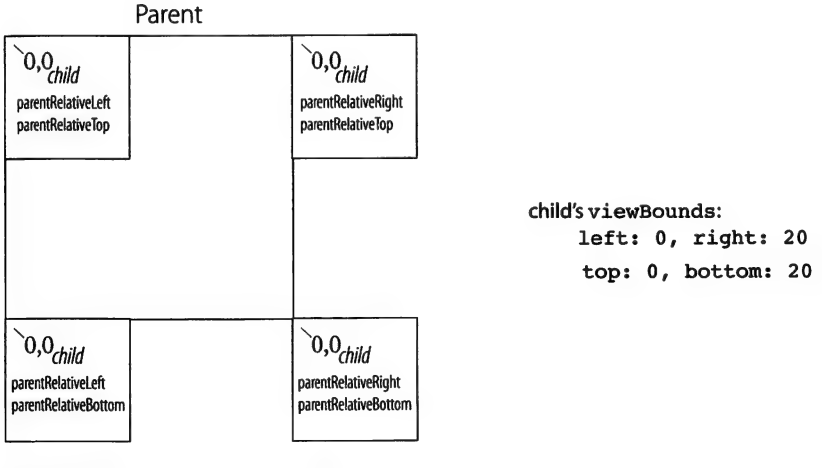


Figure 3.12 Changing the justification of a child view.

In each case, *child's* `viewBounds` remains the same (`left: 0, right: 20, top: 0, bottom: 20`). What changes is which of the parent's bounds is used as the basis for defining 0,0.

Left and Right Justification

When you use `parentRelativeLeft` justification, the values of the left and right fields of `viewBounds` are measured from the *left* of the parent (see Figure 3.13).

When you use `parentRelativeRight`, they are measured from the *right* of the parent (see Figure 3.14) and thus are usually negative.

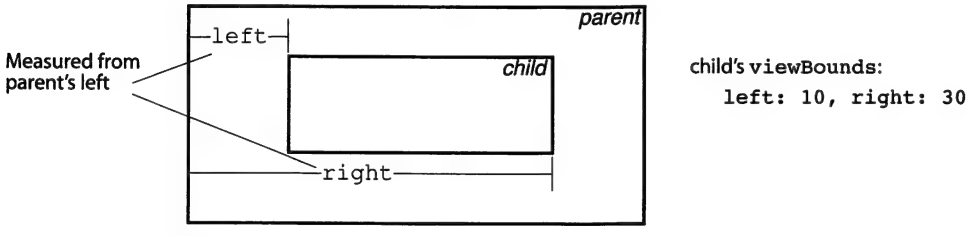


Figure 3.13 `parentRelativeLeft` justification.

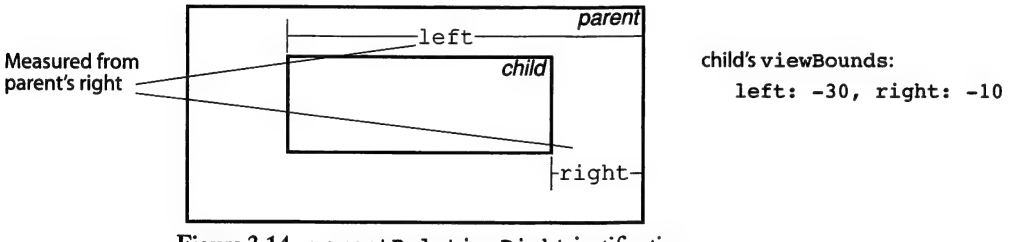


Figure 3.14 `parentRelativeRight` justification.

Top and Bottom Justification

For `parentRelativeTop` justification, the top and bottom fields of `viewBounds` are measured from the top of the parent (see Figure 3.15). For `parentRelativeBottom` justification, they are measured from the bottom of the parent (see Figure 3.16) and thus are usually negative.

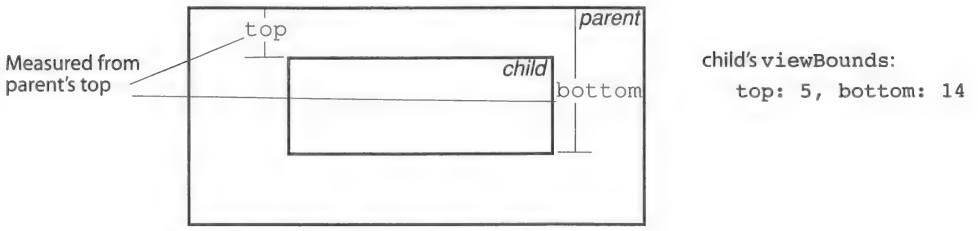


Figure 3.15 `parentRelativeTop` justification.

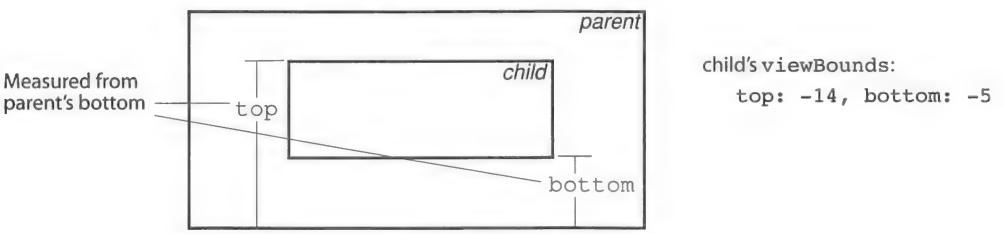


Figure 3.16 `parentRelativeBottom` justification.

You can use any combination of vertical and horizontal justification on a view—they are independent of each other. Figure 3.17 shows an example of four child views, each with different combinations of `parentRelativeTop`, `parentRelativeBottom`, `parentRelativeLeft`, and `parentRelativeRight`.

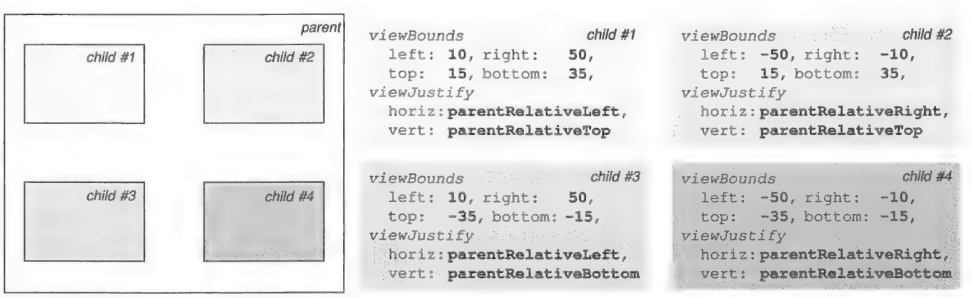


Figure 3.17 Examples of `parentRelativeLeft`, `parentRelativeRight`, `parentRelativeTop`, and `parentRelativeBottom` justifications.

Center Justification

Using center justification, you can center a view within another view. Once again, this type of justification alters the meaning of `viewBounds`. We will only look at horizontal center justification, since vertical center justification works the same way except that it uses the top and bottom coordinates instead of the left and right ones. The view system handles center justification as if the following two-step process were used:

1. First, it centers the view within its parent (see Figure 3.18).
2. Then, it adds any horizontal offset specified by the left `viewBounds` value. If the left `viewBounds` is positive, the child is offset to the *right*. If the left `viewBounds` is negative, the child is offset to the *left*. For example, if you have a child that is 20 pixels wide and you want to offset it from the center of the parent by 5 pixels to the right you would give it these `viewBounds: left: 5, right: 25` (see Figure 3.19).

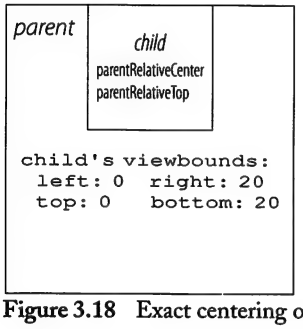
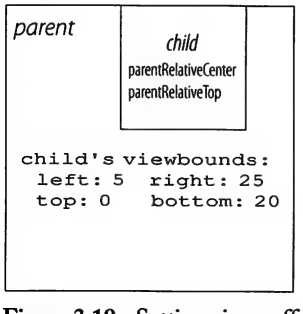


Figure 3.18 Exact centering of a view within its parent.



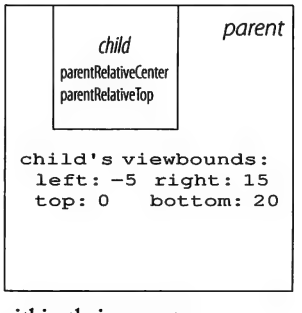


Figure 3.19 Setting views off-center by five pixels within their parents.

Full Justification

In the justifications you've seen so far, the size of the view is always fixed. In some cases, you may want a view's size to vary depending on the size of its parent. In these instances, you use `parentRelativeFull` justification (in either the horizontal or vertical direction). We will only look at horizontal full justification, since vertical full justification works the same way except that it uses the top and bottom coordinates.

When you use `parentRelativeFull` horizontal justification, the left field of the `viewBounds` specifies the offset from the left of the parent, while the right field specifies the offset from the right of the parent (see Figure 3.20). Thus, setting `left` and `right` both to 0 causes the view to be exactly as wide as its parent, while setting the `left` to 5 and the `right` to -5 causes the child to be five pixels less wide on each of its sides (see Figure 3.21).

Note: When using `parentRelativeFull` justification, `viewBounds` specifies insets or offsets—not widths. Using a positive number in the right `viewBounds` results in a child view that extends beyond its parent. Likewise, using a negative number in the left `viewBounds` makes the child extend beyond its parent.

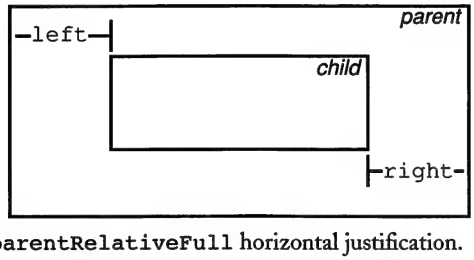


Figure 3.20 `parentRelativeFull` horizontal justification.

When you use `parentRelativeFull` vertical justification, the top field specifies the offset from the top of the parent, while the bottom field specifies the offset from the bottom of the parent. Negative values specify an offset up, while positive values specify an offset down. Setting top and bottom both to 0 causes the view to be exactly as tall as its parent.

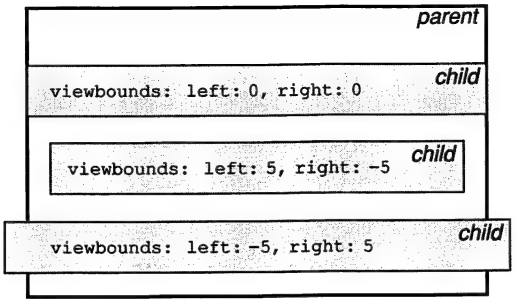


Figure 3.21 Examples of horizontal `parentRelativeFull` justification.

Uses of Parent Justification by Protos

Some system protos have specialized parent justification so that they will draw out in a particular location. For example, look at the settings of `protoCancelButton`:

`viewJustification`	`parentRelativeRight` `parentRelativeBottom`
`viewBounds`	`left`: -18, `right`: -5, `top`: -18, `bottom`: -5

Thus, a `protoCancelButton` is 13 pixels square and is located 5 pixels from the lower right edge of its parent (see Figure 3.22).

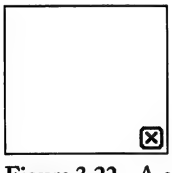


Figure 3.22 A cancel button uses parent justification to nestle in its parent's bottom right corner.

Using Sibling Justification

You are not limited to parent justification. All the same choices are available relative to a view's sibling as well. In fact, you will frequently have a number of sibling views which you will want to draw out relative to one another rather than to a par-

ent. Sibling justification provides this ability. Rather than measuring from the parent, sibling justification measures from the previous sibling.

A view's sibling is always the sibling prior to it in the NTK browser window (see Figure 3.23).

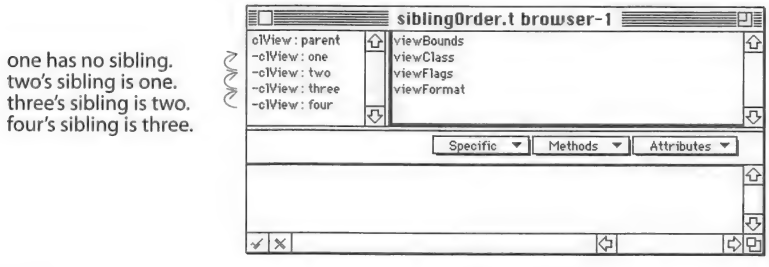


Figure 3.23 Identifying a view's sibling.

The types of sibling justification are similar to those of parent justification:

`siblingRelativeLeft` The left and right `viewBounds` slots are measured from the previous sibling's *left* `viewBounds` slot.

`siblingRelativeRight` The left and right `viewBounds` slots are measured from the previous sibling's *right* `viewBounds` slot.

`siblingRelativeTop` The top and bottom `viewBounds` slots are measured from the previous sibling's *top* `viewBounds` slot.

`siblingRelativeBottom` The top and bottom `viewBounds` slots are measured from the previous sibling's *bottom* `viewBounds` slot.

`siblingRelativeFull` For vertical justification, top and bottom `viewBounds` slots are measured from the previous sibling's top and bottom slots. For horizontal justification, left and right `viewBounds` slots are measured from the previous sibling's left and right `viewBounds` slots.

`siblingRelativeCenter`	For vertical justification, the view is centered within its previous sibling's height and then offset by the top `viewBounds` slot. For horizontal justification, the view is centered within its previous sibling's width and then offset by the left `viewBounds` slot.

Interactions between Parent and Sibling Justification

Until now you have only been able to set a view's justification relative to either a sibling or its parent. You can also set a view's sibling and parent justification simultaneously.

As you can set both parent justification and sibling justification for a view, it is important to understand what order of precedence is used. For a view that has a sibling, any sibling justification overrides any parent justification. For a view that has no sibling (a view that is the first child of its parent), the parent justification is used and any sibling justification is ignored.

Caution: If you use `siblingRelativeFull` justification you currently need to use either `parentRelativeTop` or `parentRelativeLeft` justification. If you don't, your view will not be placed in the correct position.

Example

The ability to define both parent and sibling justification can make it easier to specify one value for a whole set of views. For instance, you may want to set up a row of views that snap right next to each other. You can do this by using the following values for `viewJustify` and `viewBounds`:

`viewJustify`	vertical: `parentRelativeFull` and `siblingNone` horizontal: `parentRelativeleft` and `siblingRelativeRight`
`viewBounds`	`top`: 5, `bottom`: -5, `left`: 0, `right`: 20

You'll end up with a row of views, with the first one at the left, each twenty pixels wide, touching one another, as shown in Figure 3.24.

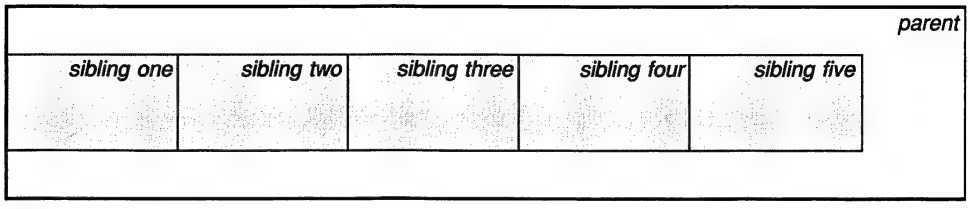


Figure 3.24 Using a combination of `parentRelativeFull` and `siblingRelativeRight` justification.

Another way to set up a row of views is to put the first one at the right, and put successive siblings to the left. You can do this by using the following values for `viewJustify` and `viewBounds`:

`viewJustify`	vertical: `parentRelativeFull` and `siblingNone` horizontal: `parentRelativeRight` and `siblingRelativeLeft`
`viewBounds`	`top`: 5, `bottom`: -5, `left`: -20, `right`: 0

You'll end up with a row of views, with the first one at the right, each twenty pixels wide, touching one another, as shown in Figure 3.25.

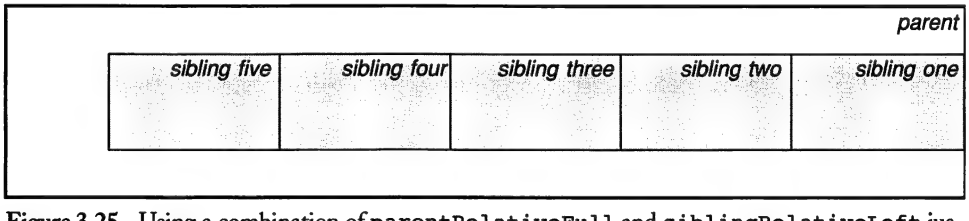


Figure 3.25 Using a combination of `parentRelativeFull` and `siblingRelativeLeft` justification.

Justifying the Application Base View

Having been told that Newton is a family of products with a variety of screen sizes, it is quite logical of you to assume that you can resize your application base view to the size of the Newton screen—larger for a larger screen and smaller for a

smaller screen. You would also want all of the child views of the base view to lay themselves out differently, becoming smaller or larger as necessary. While you can resize your application base view depending on the screen size, you don't do it using `viewJustify`. Instead, you write NewtonScript code to obtain the current screen size and set your `viewBounds` based on that screen size.

There are two reasons why this is done in code. The first is that most applications will want to set a maximum horizontal and/or vertical size. If a 2-foot by 3-foot screen size becomes available, your application will probably not want to take up all that space, but rather grow no more than a preset value.

The second reason is that for the base view, the `parentRelative` justifications don't justify based solely on the screen size; the icon bar is included also. Thus, if you wanted an application exactly as big as the screen, you might try to use `parentRelativeFull` justification horizontally and vertically, with `viewBounds` of 0 for top, bottom, left, and right.

If you do this, however, you would end up with an application view that dips below the visible screen area to fill the icon bar—certainly not what you intended. Although you could try to work around this by determining the size of the icon bar, and account for that in your `viewBounds`, there is no guarantee that the icon bar will remain its current height, or even that it will always be at the bottom of the screen. Therefore, it's smarter and safer to set your application size in code. See "Setting Application Bounds Based on the Screen Size" on page 343 for sample code that shows setting the size of your application base view.

Text and Graphics Justification

You can also use `viewJustify` to affect where graphics and text are drawn within a view. For example, you can have text within a `protoStaticText` drawn so that it is at the bottom left or the top right of a view. As another example, a picture within a `clPictureView` could be drawn at the top of the view, or so that it is centered exactly within the view.

Horizontally, your choices are:

left	Left alignment.
center	Center alignment.
right	Right alignment.
full	Stretches the graphics or text to fill the width of the view.

Vertically, your choices are:

top	Top alignment.
center	Center alignment.
bottom	Bottom alignment.
full	Stretches the graphics to fill the width of the view. Does nothing for text.

Text templates can also specify how much text is in them. Choices are:

no limit	Text can be multi-line.
One Line	One line of text only.
One Word	One word of text only.

Modifying the WaiterHelper Application

Updating the Detail Template

Let's revisit the detail template we created in the previous chapter. We can now take advantage of the features covered in this chapter to improve its design. There are a variety of changes we will make, some simple—like changing the fonts, and others requiring more time—like setting the justification.

Setting the Correct Fonts

Let us start with something simple—fonts. The items in the itemContainer shouldn't be in a bold font. To change the font, add a `viewFont` slot to the item1 layout (use the Attributes popup, the Specific popup, or the New Slot menu item). Change the font from `simpleFont9+tsBold` to `fancyFont12`. Do the same for the item2 layout.

Visual Effects When the Application Is Opened or Closed

1. To get the application to open with an iris and to close with a reverse iris, add a `viewEffect` slot to the `protoApp` template.
2. Choose iris open.

Build, download, and run the application. As you open it, it should open from the center of the screen.

Setting viewFormat for the Order Template

The items in the order template need a shadowed border around them. To create the shadowed border, do this:

1. Edit the `viewFormat` slot in the order template.
2. Set the frame to black, the pen to 1, and the shadow to 2.
3. Make the template a little narrower so that you can see the shadow.

Setting viewFormat for the Detail Template

The detail template also needs a frame. Edit the `viewFormat` slot and set the frame to black and the pen to 1.

Making a Table and Chairs

We need a table, which we'll represent as a framed rectangle. There will be chairs around the table, which will also be framed squares:

1. Draw a `clView` to the right of the numPeople template. Name it "table" and sets its frame to black with a pen of 1.
2. Around the table, create four small `clViews`. Frame them black, with a pen of 1, and name them "chair1", "chair2", "chair3", and "chair4".

Make sure the chairs are children of the table. If necessary, change the hierarchy with option-right arrow and option-left arrow. If you change the hierarchy you may need to edit the `viewBounds` slot; the values need to be relative to the table template.

3. Within the table, create a `protoLabelPicker` template from which to choose a table number.

4. Name it "tablePicker" and give it an empty `text` slot.
5. Set the `labelCommands` slot of tablePicker to: `["51", "52", "53", "54", "55", "56"]` (the table numbers).

At a later point, we'll display the number of chairs based on the number of people in the party, but for now, we'll use a fixed number—4.

Figure 3.26 shows the template in NTK so far.

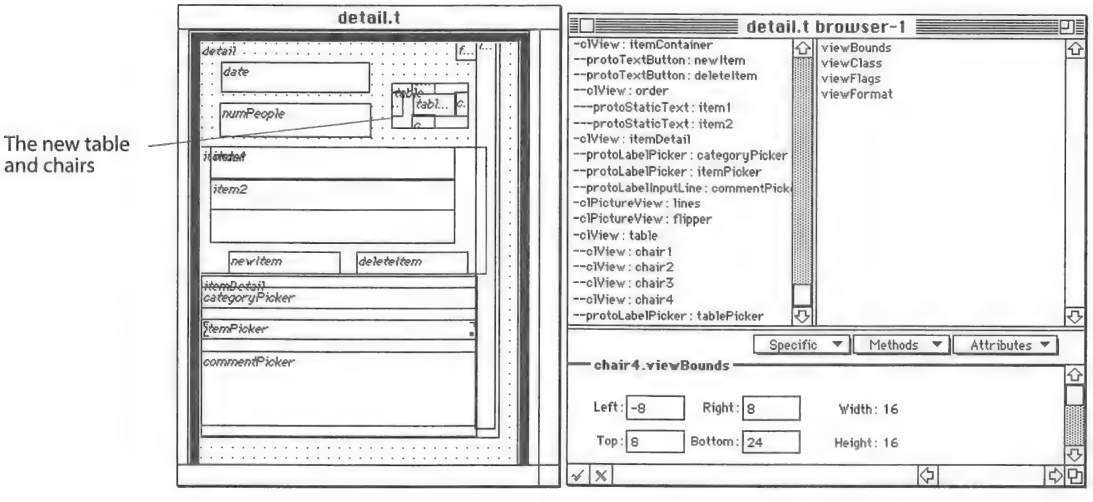


Figure 3.26 The detail template with formatting and a table.

Build, download, and run the application. Figure 3.27 shows how it should now appear.

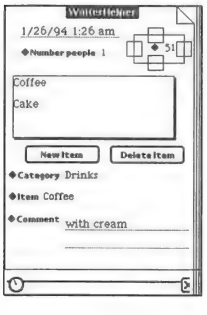


Figure 3.27 The detail view with formatting on the Newton.

Choosing Correct viewFlags

There are a few different `viewFlags` changes which need to be made:

1. To begin with, the date template needs the Field Type in `viewFlags` set to Date.

This way, double-tapping on the date view brings up a keyboard specialized for date editing (see Figure 3.28).

Figure 3.28 The date keyboard.

2. Set `vClipping` in the `viewFlags` slot of the lines template, so that the lines don't extend down into the status bar.

Figure 3.29 shows the application after these changes are made.

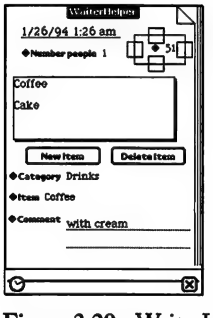


Figure 3.29 WaiterHelper after setting `viewFlags`.

Setting Justification for Each Template

It is time now for you to set the justification for your templates in the detail layout. Just remember that justification is important for handling Newtons with different screen sizes, as well as to easily set the bounds of templates relative to other templates.

Table 3.2 shows exact `viewJustification` and `viewBounds` for all the templates in the detail layout. In addition, it describes the effect desired for each template.

Template	*viewBounds*	*viewJustification*	*Explanation*
protoApp	`l:0, r:236` `t:2, b:335`	h: `parentRelativeLeft` v: `parentRelativeTop`	Temporarily same size as MessagePad/MessagePad 100 screen. For a MessagePad 110, make the bottom 319. Eventually, we'll add code to dynamically set the size based on the screen size
detail	`l:0, r: 0` `t:0, b:-25`	h: `parentRelativeFull` v: `parentRelativeFull`	Same size as protoApp, but leaving room for the status bar in the protoApp.
date	`l:5, r:-100` `t:20, b: 55`	h: `parentRelativeFull` v: `parentRelativeTop`	Expands in width if possible.
numPeople	`l: 5, r:122` `t:52, b: 73`	h: `parentRelativeLeft` v: `parentRelativeTop`	Fixed in size and position.
itemContainer	`l: 5, r:-15` `t:80, b:176`	h: `parentRelativeFull` v: `parentRelativeTop`	Expands in width if possible.
newItem	`l:-40, r:30` `t:-16, b: 0`	h: `parentRelativeCenter` v: `parentRelativeBottom`	Always at the bottom of the itemContainer, off-center to the left.
deleteItem	`l:40, r:110` `t:-16,b: 0`	h: `parentRelativeCenter` v: `parentRelativeBottom`	Always at the bottom of the itemContainer, off-center to the right.
order	`l:-2, r: -2` `t: 3, b:-15`	h: `parentRelativeFull` v: `parentRelativeFull`	Just above the newItem and deleteItem. Full width of the itemContainer.
item1	`l:0, r: 0` `t:0, b:15`	h: `parentRelativeFull` v: `parentRelativeTop`	Full width of order.
item2	`l:0, r: 0` `t:0, b:15`	h: `parentRelativeFull` v: `siblingRelativeBottom`	Full width of order, located directly below item1.
itemDetail	`l: 5, r:-15` `t:180, b: 0`	h: `parentRelativeFull` v: `parentRelativeFull`	Width of detail view. Gets taller as the detail view gets taller.
categoryPicker	`l:0, r:210` `t:5, b: 17`	h: `parentRelativeLeft` v: `parentRelativeTop`	Full justification doesn't work for `protoLabelPicker`. We'll fix it in a later chapter.

Table 3.2 Specifications for `viewBounds` and `viewJustify` in the detail template.

Template	*viewBounds*	*viewJustification*	*Explanation*
itemPicker	`l:0, r:210` `t:8, b: 20`	h: `parentRelativeLeft` v: `siblingRelativeBottom`	Full justification doesn't work for `protoLabelPicker`. We'll fix it in a later chapter.
comment-Picker	`l: 0, r:210` `t:43, b: 0`	h: `parentRelativeLeft` v: `parentRelativeFull`	Horizontal full justification doesn't work for `protoLabel-InputLine`. We'll fix it in a later chapter.
lines	`l:-9, r:5` `t:-4, b:0`	h: `parentRelativeRight` v: `parentRelativeFull`	At right of detailView, extending the full height.
flipper	`l:-25, r:-4` `t: -1, b:20`	h: `parentRelativeRight` v: `parentRelativeTop`	Located in upper right of the detail view.
table	`l:-85,r:-25` `t:41, b: 71`	h: `parentRelativeRight` v: `parentRelativeTop`	Located in upper right of the detail view.
chair1	`l: 0, r:8` `t:-4, b:4`	h: `parentRelativeCenter` v: `parentRelativeTop`	At the middle top of the table.
chair2	`l:-4, r:4` `t: 0, b:8`	h: `parentRelativeRight` v: `parentRelativeCenter`	At the middle right of the table.
chair3	`l: 0, r:8` `t:-4, b:4`	h: `parentRelativeCenter` v: `parentRelativeBottom`	At the middle bottom of the table.
chair4	`l:-4, r:4` `t: 0, b:8`	h: `parentRelativeLeft` v: `parentRelativeCenter`	At the middle left of the table.
tablePicker	`l:17, r:48` `t:10, b:25`	h: `parentRelativeLeft` v: `parentRelativeTop`	No other justification works for `protoLabelPicker`. We'll fix it in a later chapter.

Table 3.2 (continued)

Other Adjustments

There are just three more details to take care of in the detail layout.

1. So that the commentPicker doesn't draw outside of the itemDetail, set the `vClipping` flag of itemDetail.

We also need to add a label at the top of the detail template giving the current check number.

2. Create a `protoTitle` named "title" as a child template of the detail template. Give it the following slots:

```
viewBounds      {left: 0, top: 0, right: 80,
                bottom: 13}

     title      Check #683
```

Now that we have the title template at the top of the application, we no longer need the title we get from the `protoApp` ("WaiterHelper").

3. Obscure the `protoApp` title by setting the fill of the detail template to white (in `viewFormat`).

Figure 3.30 shows the application running.

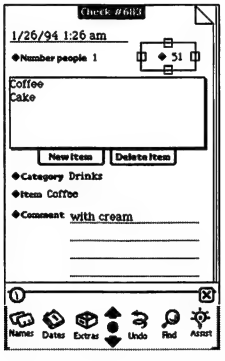


Figure 3.30 Detail view after all changes.

To verify that the `viewJustification` has been implemented correctly, change the `viewBounds` of the `protoApp` to `{left: 0, right: 216, top: 12, bottom: 325}` These numbers are for a MessagePad/MessagePad 100. For a MessagePad 100, make the bottom 309. For other Newtons, make the `viewBounds` smaller than the actual screen size.

This will cause the application view to be smaller. We can see the effect of that change correctly reflected in all the descendant views as shown in Figure 3.31. This type of testing should make you more confident that the application will run correctly on Newtons with different screen sizes. Make sure to restore the `viewBounds` to the original value before going any further.

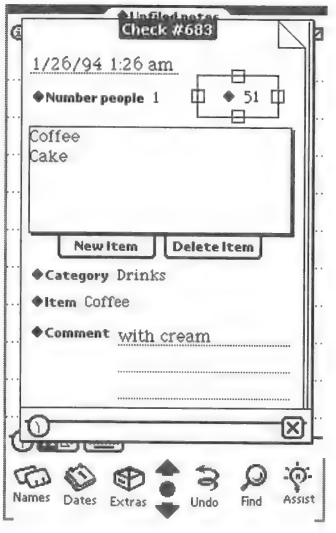


Figure 3.31 The application after shrinking its size.

Creating the Overview

Now that we've got the detail template whipped into shape, we need to work on the overview template. Figure 3.32 shows the results we are looking for.

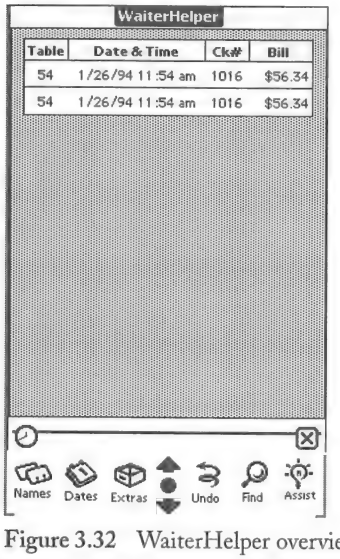


Figure 3.32 WaiterHelper overview on the Newton.

Creating the Layout, Overview Container, and Header

1. Create a layout window, name it "Overview.t", and add it to the project.

The overview is composed of the header, which gives labels for each column, and the area used for the actual rows. We'll group the text for the header in one template, and we'll group all the rows into another template. Since a layout can have only one topmost template, we'll have to put the header template and the rows template inside some other containing template. Figure 3.33 shows the structure of the overview as well as the template names we'll use.

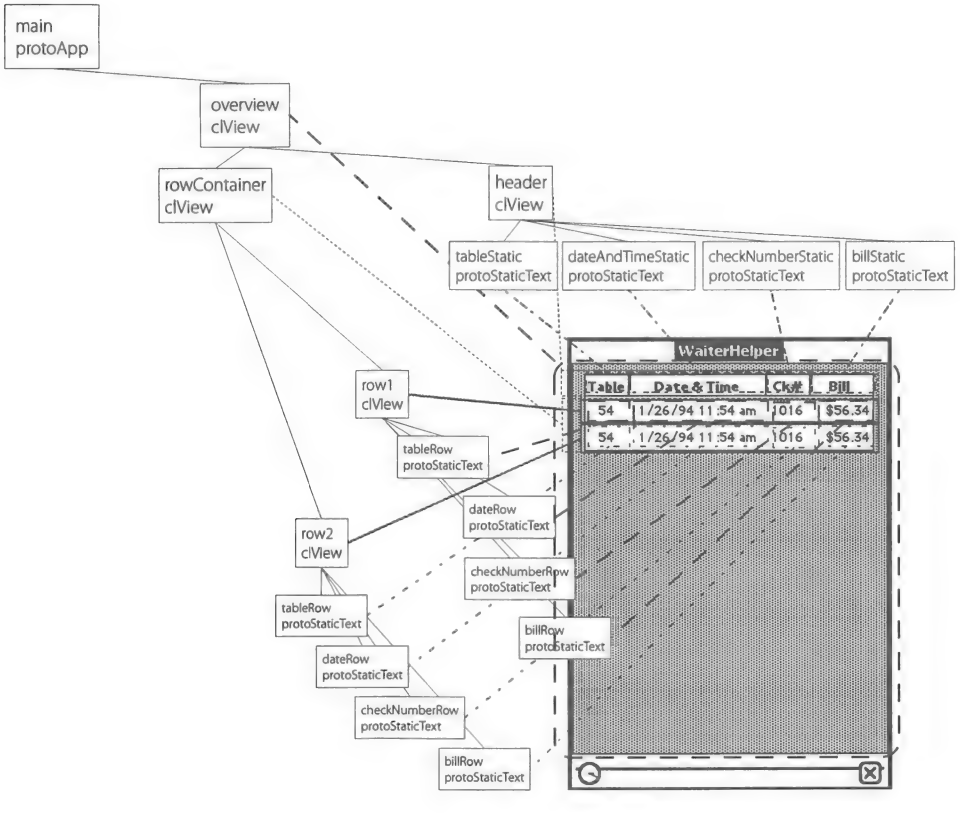
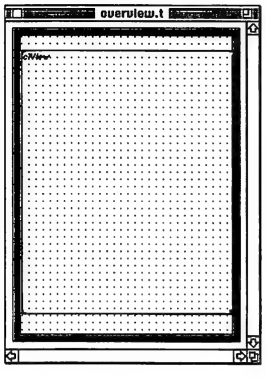


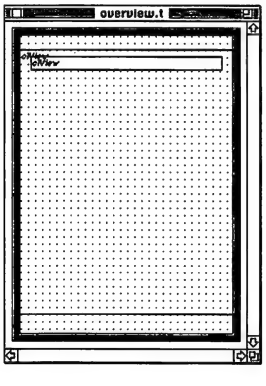
Figure 3.33 The hierarchy of templates in the overview layout.

2. Draw a `clView` in the overview.t layout named "overview" with the following characteristics:



viewBounds	left: `0` right: `0` top: `16` bottom: `-25`
viewJustify	horizontal: parent: full, sibling: none vertical: parent: full, sibling: none
viewFormat	pen: `1` frame: `black` fill: `light gray`

3. Draw another `clView` at the top of overview; name it "header". Set the following slots:

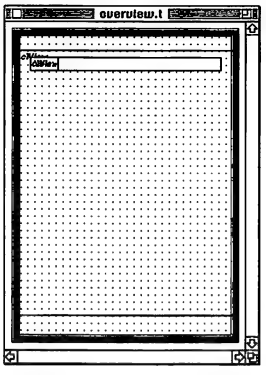


viewBounds	left: `0` right: `216` top: `08` bottom: `24`
viewJustify	horizontal: parent: center, sibling: none vertical: parent: top, sibling: none
viewFormat	pen: `1` frame: `black` fill: `white`

The header won't resize horizontally if the application gets bigger. Instead, it will stay centered within the overview. The user will just see more light gray.

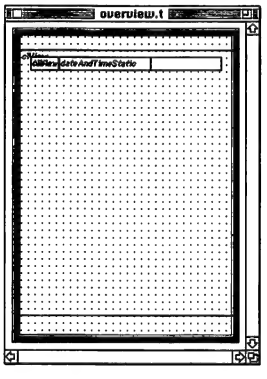
4. Within the header, draw four `protoStaticTexts`. Name them "tableStatic", "dateAndTimeStatic", "checkNumberStatic", and "billStatic". Now adjust each of their slots accordingly.

Set tableStatic as follows:



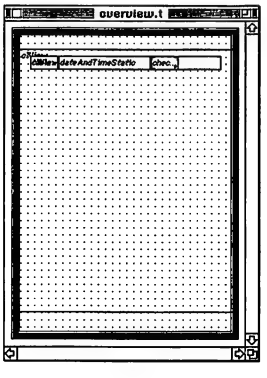
text	`Table`
viewBounds	left: `0` right: `32` top: `0` bottom: `0`
viewJustify	horizontal: parent: left, sibling: none vertical: parent: full, sibling: none text/graphics horizontal: center text/graphics vertical: `center`
viewFormat	pen: `1` frame: `black`

Set dateAndTimeStatic as follows:



text	`Date & Time`
viewBounds	left: `0` right: `104` top: `0` bottom: `0`
viewJustify	horizontal: parent: left, sibling: right vertical: parent: top, sibling: full text/graphics horizontal: `center` text/graphics vertical: `center`

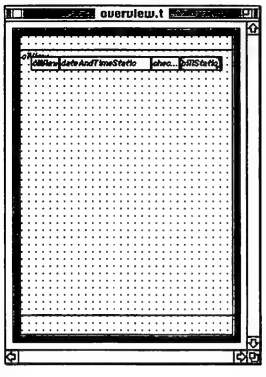
Set checkNumberStatic as follows:



text	`Ck#`
viewBounds	left: `0` right: `32` top: `0` bottom: `0`
viewJustify	horizontal: parent: left, sibling: right vertical: parent: top, sibling: full text/graphics horizontal: `center` text/graphics vertical: `center`

viewFormat	pen: `1` frame: `black`

Set billStatic as follows:



text	`Bill`
viewBounds	left: `0` right: `46` top: `0` bottom: `0`
viewJustify	horizontal: parent: left, sibling: right vertical: parent: top, sibling: full text/graphics horizontal: `center` text/graphics vertical: `center`

Note that the static texts are justified—all are the same height as the first and each starts horizontally where the previous one ends. The first and third are framed so that there is a vertical line between each header.

At this point, you can try a preview from NTK to get an idea of what the overview will look like (see"The Layout Menu" on page 353). Figure 3.34 shows just such a preview—but notice that it is not perfect (sort of like cubic zirconium).

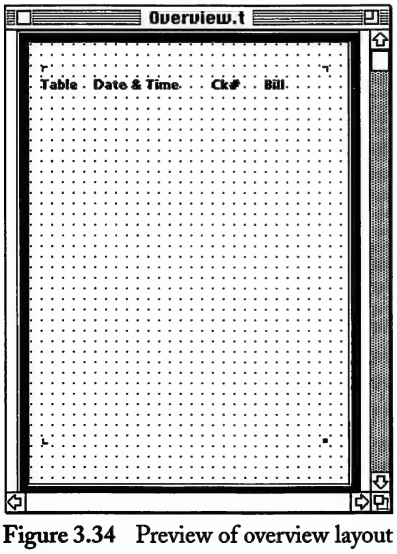
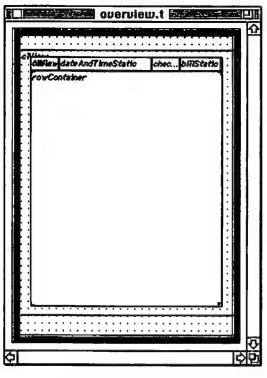


Figure 3.34 Preview of overview layout with header complete.

Creating the Row Container and Two Rows

1. Start by drawing a `clView` below the header. Name the template "rowContainer" and set the following slots:



viewBounds left: `0`
right: `0`
top: `24`
bottom: `-8`

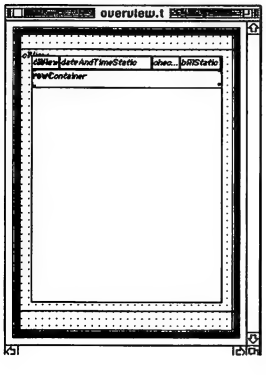
viewJustify horizontal: parent: left, sibling: full
vertical: parent: full, sibling: none

The rowContainer will grow taller if the application becomes larger, thereby accommodating more rows. Horizontally, rowContainer is the same size as the header and is always centered within the application.

Note: Since rowContainer does not adjust depending on the width of the application, the application width cannot be thinner than either the header or the rowContainer (i.e., 216). It is bad manners to have your main view cut off the edges of other views.

---

2. In rowContainer, draw a `clView` and name it "row1". Set its slots as follows:



viewBounds left: `0`
right: `0`
top: `1`
bottom: `20`

viewJustify horizontal: parent: full, sibling: none
vertical: parent: top, sibling: none

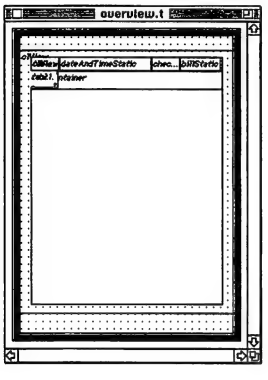
viewFormat pen: `1`
frame: `black`
fill: `white`

The row1 template will draw in white (obscuring the light gray of the overview) and will frame in black.

3. Within row1, draw four `protoStaticTexts`, each as wide as the corresponding static texts in the header. Name them (from left to right) "tableRow", "dateRow," "checkNumberRow", and "billRow".

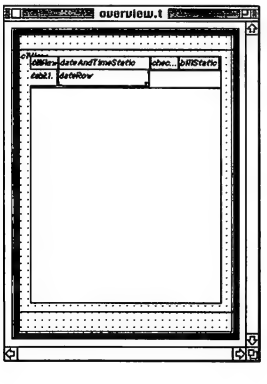
4. Now, set each of their slots as follows.

Set the tableRow slots to:



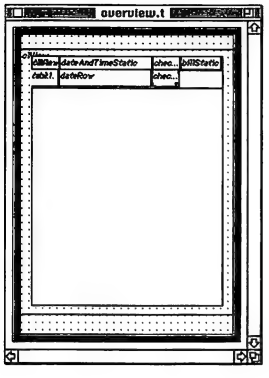
text	`54`
viewBounds	left: `0` right: `32` top: `0` bottom: `0`
viewJustify	horizontal: parent: left, sibling: none vertical: parent: full, sibling: none text/graphics horizontal: `center` text/graphics vertical: `center`
viewFont	`simpleFont9`

Set the dateRow slots to:



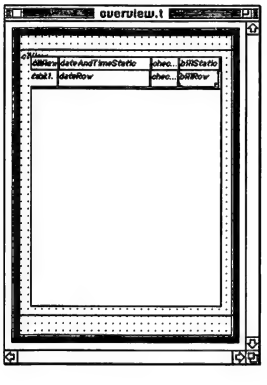
text	`1/26/94 11:54 am`
viewBounds	left: `0` right: `104` top: `0` bottom: `0`
viewJustify	horizontal: parent: left, sibling: right vertical: parent: top, sibling: full text/graphics horizontal: `center` text/graphics vertical: `center`
viewFont	`simpleFont9`

Set the checkNumberRow slots to:



text	`1016`
viewBounds	left: `0` right: `32` top: `0` bottom: `0`
viewJustify	horizontal: parent: left, sibling: right vertical: parent: top, sibling: full text/graphics horizontal: `center` text/graphics vertical: `center`
viewFont	`simpleFont9`

Set the billRow slots to:



text	`$56.34`
viewBounds	left: `0` right: `46` top: `0` bottom: `0`
viewJustify	horizontal: parent: left, sibling: right vertical: parent: top, sibling: full text/graphics horizontal: `center` text/graphics vertical: `center`
viewFont	`simpleFont9`

Note that each of the static texts starts at the right end of the previous. The text is not bold. Rather than re-entering another row, select the row1 template (double-click on row1 in the browser window to select).

5. Copy row1, select the rowContainer, and then paste.

6. Select the new row from the browser window, name it row2, and set its vertical `viewJustify` to `siblingRelativeBottom` so that it appears below the first row.

Linking and Building

Before building the application, link the overview layout to the main layout. Figure 3.35 shows what the main layout should look like after you've set up the linked layouts.

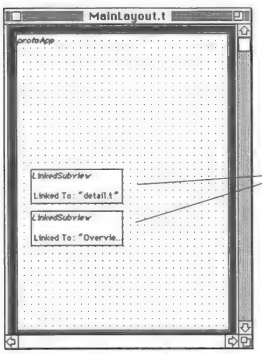


Figure 3.35 Main layout after linking to "overview.t".

If you were to build your application now, both the overview and the detail would display. Since we want to look at the overview, edit the `viewFlags` of the detail template to turn off `vVisible`—with the click of a button it disappears. Now, build, download, and run the application (see Figure 3.31).

Summary

In this chapter, you learned about the various slots in a view that determine its look and behavior. You also learned how to use parent and sibling justification to modify the position and size of the view. You then saw how we modified the WaiterHelper application to take advantage of the various view settings and justifications that we had just described. Finally, you added another layout window to the application to handle the overview.

Chapter 4

Protos

I have also seen children successfully surmounting the effects of an evil inheritance. That is due to purity being an inherent attribute of the soul.

—*Mahatma Gandhi*

Protos are the Newton's solution to tight memory constraints. They are also the vehicles of code reusability and thus the components of code libraries. The definition of a proto is fairly straightforward:

- It is just a *predefined template* containing certain visual characteristics and particular behavior.

When we speak of a template protoing from a proto, we mean that it inherits its initial visual characteristics and behavior from that proto.

It is quite similar to an ordinary family structure. You look and act in many ways like your mother and father, but are seldom carbon copies of them. Likewise,

if you have sisters or brothers, they are rarely identical twins. Similarly, a template will usually have some additional elements added to it that make it look or act differently from both its proto and other templates that proto from the same proto.

Note: Yes, *proto* is both a noun and a verb. We will commonly speak about the "proto of something" as well as "the act of protoing." Here is a weird sentence that you will grow quite accustomed to: a template protos from its proto.

Introduction to Protos

Consider a button on the Newton, as a perfect proto candidate. Everything from its visual characteristics—its rectangular frame, rounded corners, and text string (like "Press Me")—to its particular behavior—it does something when you tap on it—screams for protoing.

The Newton design team wisely decided that you should not have to reinvent this type of button, so they provided protos and in this case a protoTextButton. When you want a button in an application, you just draw out a template that protos from protoTextButton and you get all of the proto's button characteristics without any extra work. Your job as the programmer is to add the specialized components: what happens when you tap on the button, where the button is located, and so on. Further, because a great number of protos have been built into the Newton ROM, you get to use them without paying the price in much additional memory space. And just in case the Newton designers forgot some really useful proto on the ROM, they gave you the ability to design your own custom protos.

But enough fanfare, let's get down to the mechanics of protos, how they work, how you use them, and how you create them.

Kinds of Protos

So, a proto is simply a predefined template (just a particular type of NewtonScript frame) that provides a desired appearance and behavior. The two kinds of protos are also referred to in different ways. The ones in the Newton ROM are called *system protos* and the ones you create are called *user protos*. Besides differences in

naming, there are also some differences in how you create user protos and how you manage them inside your application project. (The relevant differences will be explained when you learn how to create you own protos.)

Note: Don't be confused by the name *user proto.* The user in this case is you, the NTK programmer, and not the Newton owner you think of as a user. Remember, after all, you are users to the NTK programming team.

Proto Nuts and Bolts

It is quite simple to use protos in an application. During design, you select a proto in the NTK tool palette and then draw out a corresponding template in a layout window (see Figure 4.1). When you create this type of template, NTK adds a `_proto` slot to it (see Figure 4.1) that contains a pointer to the system or user proto it is based upon. At run time, the view that is created from the template inherits all of the proto's slots. These slots contain data values and/or methods from the proto via this pointer (see Figure 4.1). Inheritance is covered in detail in "Proto Inheritance" on page 152.

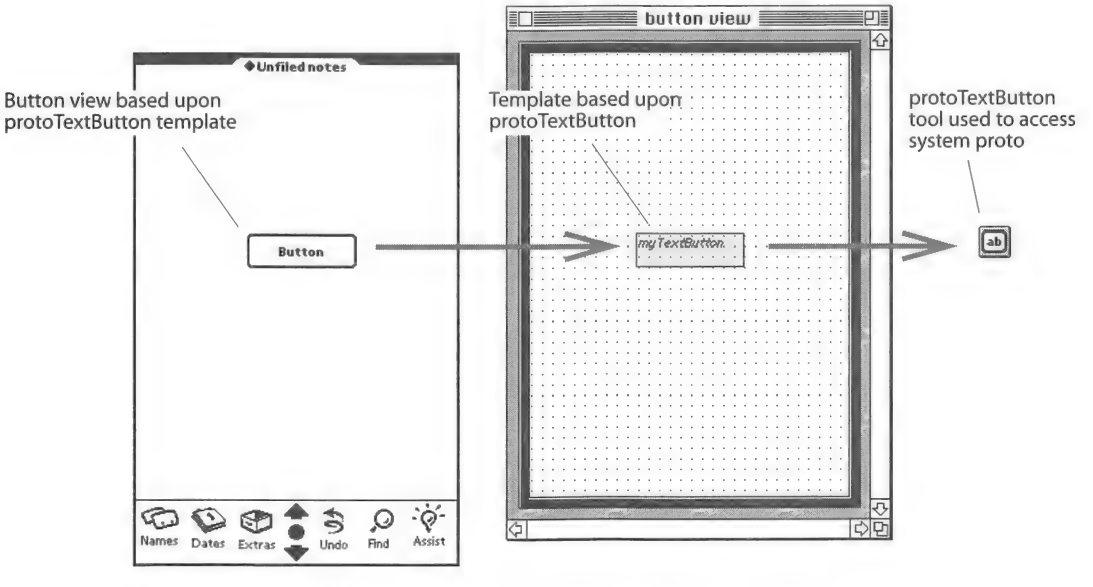


Figure 4.1 Creating a view based upon a template based upon a proto.

NTK takes care of adding the proper slots to your template automatically. But you should remember that whether you use standard NewtonScript frame syntax or NTK's graphic slot editor, the effect of creating templates based on protos is the same. As you can see in Figure 4.2, the `protoTextButton` template, named myTextButton, contains four slots.

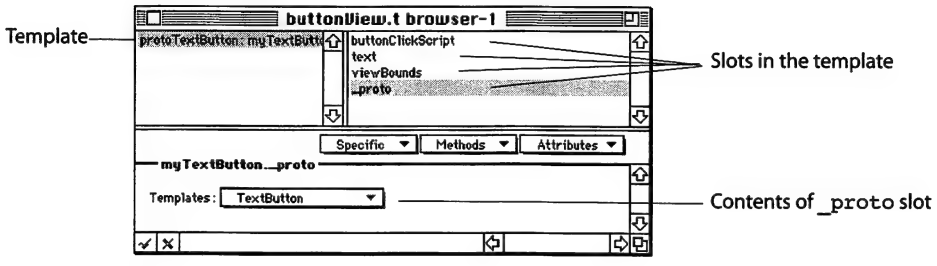


Figure 4.2 A template based on a proto contains a `_proto` slot pointing to the proto.

Notice in Figure 4.2 that the _proto slot contains a reference to `protoTextButton`, the proto upon which it is based. You could also portray this same template in standard NewtonScript:

```
MyTextButton := {
   buttonClickScript: …,
   text: "Button",
   viewBounds: …,
   _proto: protoTextButton
}
```

Protos and Template Size

The above template, myTextButton, is a frame containing four slots. Its size is based upon adding the values found in the first three slots, and adding another four bytes for the pointer to `protoTextButton`. Because of the way inheritance works, the template gets the benefit of protoing from `protoTextButton`, without having the pay a price in memory size. Thus, even though the original `protoTextButton` is a much larger structure, the template remains small in size (see Figure 4.3).

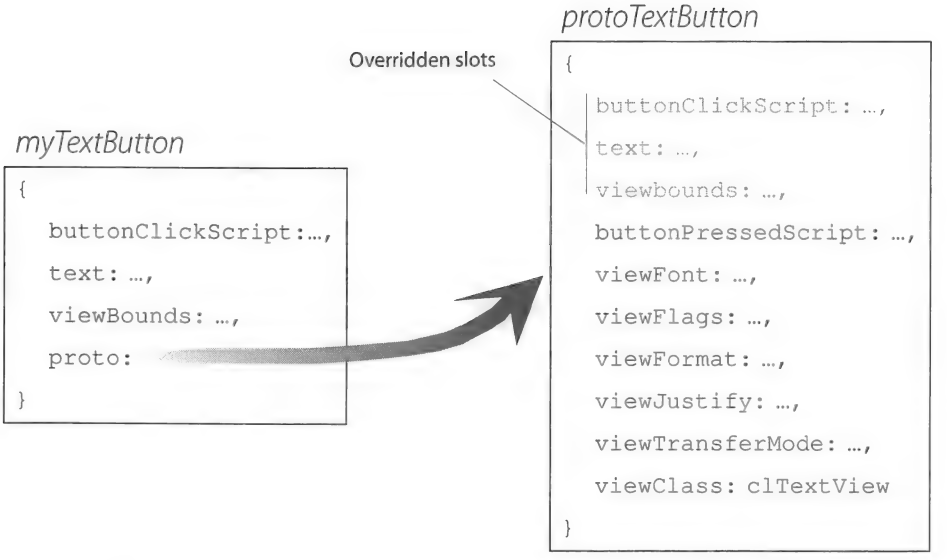


Figure 4.3 The relationship between a template and its proto.

Note: In C++, an object that inherits from another object gains not only functionality but size. In C++, a 4-byte object inheriting from a 100-byte object ends up 104-bytes big. So in NewtonScript, how big would a 4-byte object protoing from a similar 100-byte object be? The answer is eight bytes (four belong to the original template and another four to store the `_proto` pointer). The speed, versus size tradeoff was made in favor of size.

Protos Provide Template Reuse

In the discussion of linking templates (see "Linking Templates" on page 31), you learned that linking doesn't provide a way to reuse templates, since a layout can only be linked once. On Newton, protos are the mechanism for reuse. Anything that you want to use in multiple templates, be it functionality or visual appearance, you can capture in a custom user proto. Protos provide for reuse of code in much the same way that libraries operate in many other programming environments.

This should also lead you to a clearer understanding of application design on the Newton. You do not have multiple templates with identical slots; rather you gather common slots together into a proto. Then, you use such protos in your templates. *You capture uniqueness in your templates and commonality in your protos* (see Figure 4.4).

This is true of complex functionality as well as more superficial aspects like appearance. For example, we use a proto to handle the visual display and data manipulation of rows in the overview.

Now, consider a simpler case. In Figure 4.4, there are three templates that have several duplicate slots and some unique slots. Each template creates a view where the user can write a list of items (groceries, errands, or chores). Instead of creating three similar templates, the proper approach would be to create one user proto that contained all of the common slots. That user proto would be referenced by each template via a proto slot. In the templates, you would set unique elements, like the screen location of the view and the list label: chores, errands, or groceries.

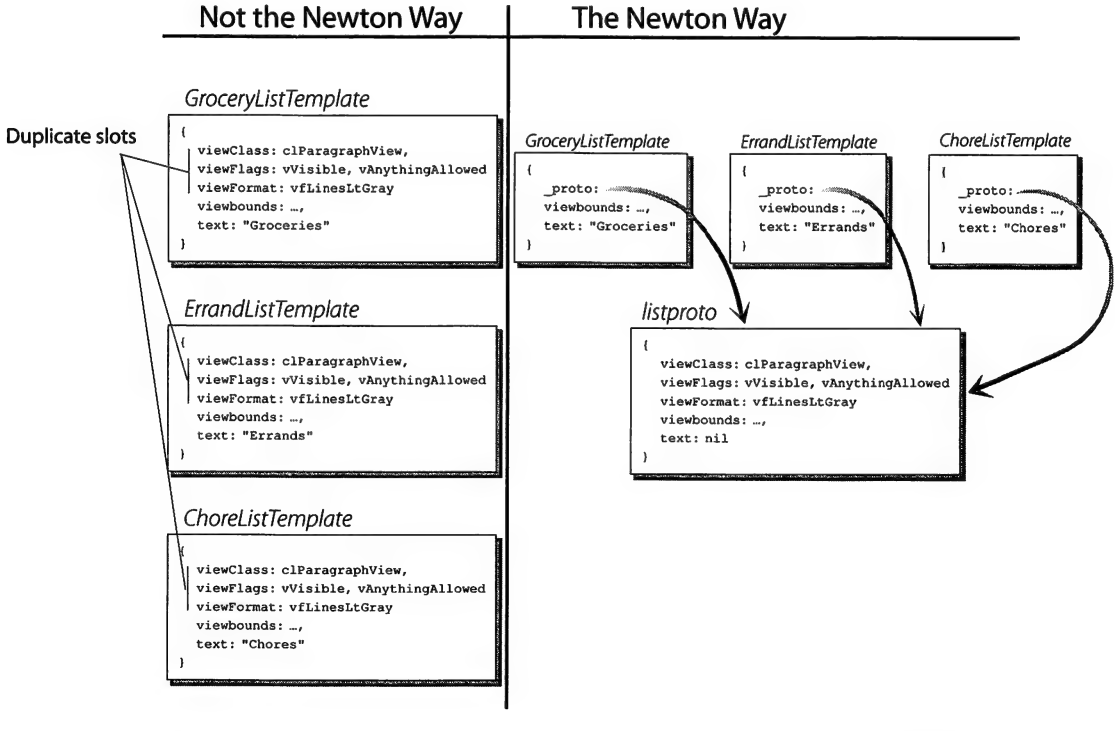


Figure 4.4 Creating a user proto to capture commonality in a number of templates.

Protos Reduce Application Size

Using protos—instead of multiple templates with duplicate functionality—reduces the total size of the application. System protos are even better in that they require no additional space in your application because they are in ROM. User protos still save space because they help reduce the size of the template in which they are used.

Reducing application size is especially important for an application that does not come on a PCMCIA card. A large application uses up limited memory on a Newton (either internal memory or PCMCIA RAM card memory). Such memory is an extremely precious resource, and your users will begrudge you every byte, so use it carefully. In addition, Newton Connection takes longer to synchronize as an application gets bigger.

Protos Increase Maintainability

Maintaining duplicates of information that should be the same is always risky. At application revision time, it can be very difficult to remember everywhere that the duplicates appear. Failing to update all copies, and thereby causing inconsistencies, is the motivating force behind programmers using symbolic constants (like `kNumberSecondsPerDay`) rather than manifest constants (like `1440`). Having to change information in only one location, rather than in multiple locations, reduces program maintenance errors.

By using protos, there is no fretting over multiple copies of information. You update slots in one location (the proto), rather than in multiple locations (a number of templates). Then all that remains is to rebuild the project in which the revised proto is used.

Protos Can Be Shared between Projects

Another benefit of protos is their reusability in multiple applications. Protos enhance Newton's already fairly quick development cycle. With protos, the second Newton application you write can take advantage of the first one. If you put your particularly nice buttons, snazzy overview entry design, or input lines with a dynamic picker into protos, you will be able to use them in other applications. Protos also provide a way to have consistency of design or behavior across a number of different applications.

To use protos in many applications you just keep them stored in a central library location and add them to each application project as needed. However, NTK currently requires you to copy the shared proto file rather than keeping a single copy.

User protos, unlike system protos, that are reused in multiple applications do not further reduce application size, however. Each application makes its own copies of its protos at build time. Thus, sharing user protos among applications is not a way to reduce the size of applications. Nevertheless, the other benefits still apply.

The System Protos

There are 10 view classes on which the system protos are based. From those, the Newton designers created 46 system protos for you to use in NTK. Some of these protos you will use all the time, like `protoStaticText`, and some only occasionally, such as `protoLabeledBatteryGauge`.

You can select the system protos from the NTK tool palette or from the pickable list. Once you select the proto and draw out the template, NTK takes care of adding the slots to the template that you need to set. System protos have some slots that you can override as you wish and other slots that you should not override.

It is also possible to create views dynamically that are based on system protos. When you do this in code, you will need to refer to the NTK documentation on each system proto to know which of the proto's slots you should override and which ones you should inherit untouched.

Table 4.1 contains a complete list of the currently available system protos. Many have a brief description but in most cases the name alone is just as good an indicator of what the proto does. In looking over these protos, you should also note that some protos require the use of other protos. For instance, a set of radio buttons should all be children of a parent template that protos from `protoRadioButtonCluster`.

protoApp—used to create an application view. There should only be one per project.	**protoCloseBox**—standard view close box. A picture button containing an 'X' icon.	**protoDateExpando**—expandable text edit line for dates.	**protoTable**—creates one-column table of text. Each table item is tappable.
protoFloater—creates a floating view, above other views.	**protoCheckBox**—for a checkable box, with the label to the right.	**protoPhoneExpando**—expandable text line for phone numbers.	**protoTableEntry**—one of the entries in a protoTable.
protoFloatNGo—same as protoFloater with a close box.	**protoRCheckBox**—for a checkable box, with the label to the left.	**protoFilingButton**—button to change an entry's folder.	**protoSetClock**—creates a clock that you can set.
protoInputLine—allows the user to input one line of text.	**protoRadioButton**—one of the radio buttons in a radio button cluster.	**protoShowBar**—picker to choose a folder to display.	**protoPicker**—creates a picker with some items to pick from.
protoLabelInputLine—for one-line input with a text label or a optional picker.	**protoPictRadioButton**—one of the picture radio button views in a radio button cluster.	**protoLabelPicker**—for a label with a text item next to it. Label is tappable for popup picker.	**protoPictIndexer**—displays an array of pictures. Any picture can be selected.
protoGauge—creates a read-only gauge.	**protoRadioCluster**—used to group exclusive choice radio buttons.	**protoRollBrowser**—like a protoRoll except it is a self-contained application.	**protoDrawer**—who knows?
protoSlider—creates a gauge view that the user can set.	**protoStaticText**—one line of read-only text. This can be a label or a tappable entry.	**protoRoll**—creates a paper roll-like view including other views that display as one line or as full views.	**protoPrintFormat**—for printing and faxing.
protoBorder—view filled with black that can be border.	**protoGlance**—creates a text view that closes after opening briefly.	**protoRollItem**—one of the views in a protoRoll or protoRollBrowser.	**protoKeyboard**—creates a floating keyboard view that is draggable.
protoDivider—creates a divider bar the width of its parent view consisting of boxed text that is at the left of the line.	**protoStatus**—creates a status bar at the bottom of a view including the clock and cancel button.	**protoActionButton**—picker of various user actions including printing, mail, and duplicate and delete.	**protoKeypad**—sets default key characteristics for a keyboard view.
protoTextButton—round rectangle button with one line of text inside.	**protoStatusBar**—creates a status bar with a clock at the bottom of a view.	**protoTitle**-creates a title centered in a black round rectangle at the top of a view.	**protoLabeledBattery-Gauge**—creates a battery gauge with a label.
protoPictureButton—icon button, tapping causes action.	**protoExpandoShell**—shell view that contains protoTextExpando child views.	**protoOverview**	**protoRecToggle**—the button that toggles the text and graphics recognizers.
protoCancelButton—picture button with an 'X' icon that is bigger than the protoCloseBox.	**protoTextExpando**—expandable text line that is dynamically displayed with a user tap.	**protoTextList**—list of text items.	

Table 4.1 The system protos.

Some Useful System Protos

You will use a handful of the available system protos quite often in a typical application. Here is a summary of some of the more common uses and some recommendations on which protos to use for particular jobs:

Unchanging Text	A `protoStaticText` provides read-only text. Your application can change the text if it wishes, but it is not directly editable by the user. Just as with any template, you can set the `vClickable` view flag to receive taps.
User Text Entry	Usually `protoInputLine` will suffice for most simple input areas. You can make these protos longer than one line. If you have many text entry areas in one view (like the Names application), consider using `protoRollItem`.
Pickable Lists	Use `protoLabelPicker`. If the user can edit the item in the list, then you should use `protoLabelInputLine`.
Container Views	Use `clView` in most cases as it is more generic than anything in the plain wrap section of the supermarket.
Picture Views	Use `clPictureView` as it has an `icon` slot to hold the PICT resource.
Buttons	Use `protoTextButton` for text buttons and `protoPictureButton` for picture buttons.
The Application View	Use either `protoApp` or `clView` with a `protoStatus`. If you use `clView`, make sure to set the `vApplication` bit in the `viewFlags` slot and add a `declareSelf` slot with the value `'self`.

Configuration of Some System Protos

Now let us explore the makeup of a few of the system protos. From this analysis, you should get a better understanding of how to build a proto, what kind of func-

tionality it makes sense to put in protos, and some of the visual aspects you can modify. It will also introduce you to some of the more common slots found in system protos.

protoPictureButton

A `protoPictureButton` contains the following slots and values:

`viewClass`	`clPictureView`. The `clPictureView` looks for an `icon` slot containing a picture or bitmap. If no slot is found, no picture is drawn. Templates that proto from `protoPictureButton` should define an `icon` slot.
`viewBounds`	`NIL`

`viewFlags`	`vVisible`
	`vReadOnly`
	`vClickable`

`viewFormat`	fill: white frame: black pen: 2 roundness: 4
`viewJustify`	horizontal: `parentRelativeLeft` and `siblingNone` vertical: `parentRelativeTop` and `siblingNone` text/graphics horizontal: center text/graphics vertical: center
`buttonClickScript`	By default, does nothing; override to handle taps.
`viewClickScript`	Calls `buttonClickScript`. *Do not override this.*

protoCloseBox

A `protoCloseBox` contains the following slots and values:

`_proto`	`protoPictureButton`
`viewJustify`	horizontal: `parentRelativeRight` and `siblingNone` vertical: `parentRelativeBottom` and `siblingNone`

viewBounds	{left: -14, right -1, top: -14, bottom: -1}
viewFormat	fill: none frame: none
icon	☒
buttonClickScript	Sends base:Close() message.

protoApp

A protoApp contains the following slots and values:

viewClass	clView
viewBounds	NIL
viewFormat	fill: white frame: black pen: 1 inset: 1 shadow: 1
viewFlags	vApplication
viewJustify	horizontal: parentRelativeCenter and siblingNone
declareSelf	'base

The protoApp contains two children, a protoTitle and a protoStatus.

Creating and Using User Protos

As useful as the system protos are, there will still come a time when you want to create your own user protos to do particular jobs. You create them in special proto layout files in NTK. Once completed, you add them to your project like other layout files and they are available for selection from the same tool palette as system protos (albeit in a different list). For information on the actual mechanics of creating a user proto from within NTK, see "Creating a User Proto" on page 361. Just as with any other proto in the palette, you can select it and then draw out a template in a layout window that protos from the user proto.

How NTK Handles System Protos

There are some differences in how NTK treats system versus user protos. NTK understands a great deal more about system protos and knows to add particular slots to a template based on them (for instance, it adds `viewBounds` to a `protoStaticText` template).

For user protos, however, NTK is necessarily a bit brain dead. When you create a template that protos from a user proto, it has only one slot in it: the `_proto` slot. NTK does not add any other slots. This difference is quite important.

Let's take the example of the `viewBounds` slot to help illustrate the point. NTK adds a `viewBounds` slot to many templates protoing from system protos, for example, a `protoStaticText`. Based on what it knows about `protoStaticText`, NTK assumes (quite reasonably) that you want each `protoStaticText` template to have a different location and size (see Figure 4.5).

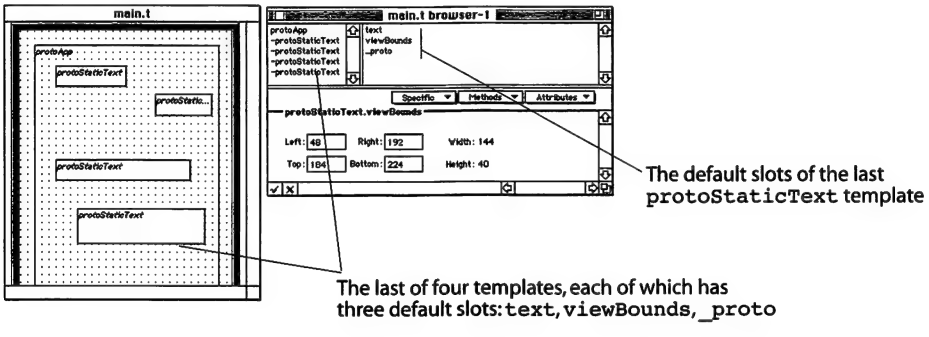


Figure 4.5 Four `protoStaticText` templates and their default slots.

On the other hand, NTK does not add a `viewBounds` slot to templates protoing from `protoStatus`. NTK makes the assumption (again, quite reasonably) that you will only have one `protoStatus` inside a particular parent view at any one time. NTK thus creates `protoStatus` templates that directly inherit the `viewBounds` slot from `protoStatus`. Each template benefits in that its location is ideally placed relative to its parent. It also means, however, that each consecutive `protoStatus` template has the same size and screen location. Since each inherits the same `viewBounds`, each is drawn right on top of the last one (see Figure 4.6).

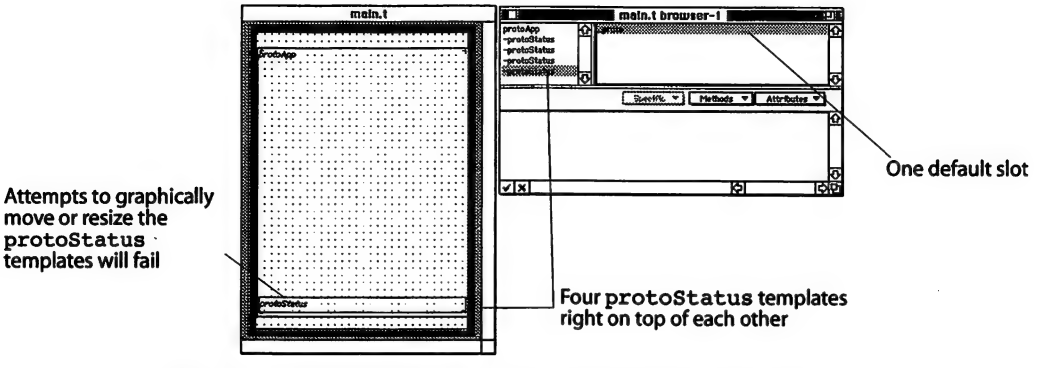


Figure 4.6 Four `protoStatus` templates and their default slots.

How NTK Handles User Protos

NTK has no way of knowing what kind of custom user proto you are creating. It can't read your mind and know whether this is the type of proto that should have its `viewBounds` slot overridden or not. Since NTK can't know, it simply doesn't do anything—no slots are added to templates protoing from user protos. Thus, all user protos inherit all of their slots (except the `_proto` slot) from the proto. You simply have to take the additional step and add a `viewBounds` slot into the templates you want to move or resize.

However, if the user proto has no `viewBounds`, NTK doesn't know where to display the template. It therefore adds a `viewBounds` to the template to deal with the situation.

Note: Any template without its own `viewBounds` slot can't be moved or resized graphically. It does not matter whether it protos from a system or user proto. You can even mimic this effect with templates based upon `protoStaticText`. Just remove the `viewBounds` slot from the template; like magic you can no longer move it around.

Referring to User Protos from Your Code

User proto templates are available as variables in your code. The variable name is the name of your proto file prefixed with `pt_`. So, if your code needs to reference a `foo` proto, it uses the variable `pt_foo`.

Protos in WaiterHelper

There are a number of ways in which we can use custom protos in WaiterHelper:

- To incorporate custom versions of `protoLabelPicker` and `protoLabelInputLine` that correctly handle horizontal full justification
- To capture the commonality of the items in the order template
- To capture the commonality of the chairs in the detail view
- To capture the commonality of the rows in the overview

Let's look at each in turn.

Using Custom Protos to Fix Problems with System Protos

The disk provided with this book (in the Custom Protos folder) contains two protos: `justifiableLabelPicker` and `justifiableLabelInputLine`. Unfortunately, the system protos `protoLabelPicker` and `protoLabelInputLine` don't correctly handle horizontal full justification. So, rather than live without full justification (after all, Newton is a family of products), we have created these two custom protos to fix that problem—an example, perhaps, of children successfully surmounting the effects of an inheritance.

To use them, copy them into your project folder and add them to your project. We will be replacing the categoryPicker, itemPicker, and commentPicker protos with these new custom protos.

1. Edit the `_proto` slot of "categoryPicker" and "itemPicker" changing each to `justifiableLabelPicker` (see Figure 4.7).
2. Edit the `_proto` slot of "commentPicker" changing it to `justifiableLabelInputLine` (see Figure 4.7).

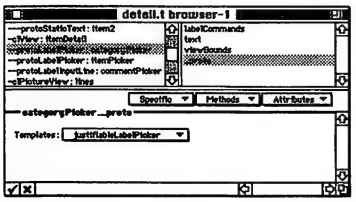


Figure 4.7 Choosing a new proto with the `_proto` slot editor.

3. Modify each of the three pickers to use full justification. Set the horizontal `viewJustify` to `parentRelativeFull` and `siblingNone`. Set the `viewBounds` left and right to 0.

With those changes in place, the picker should correctly resize horizontally.

Creating an Item Proto

Next, let's create a proto for items in the order template. When you are done it should look like Figure 4.8.

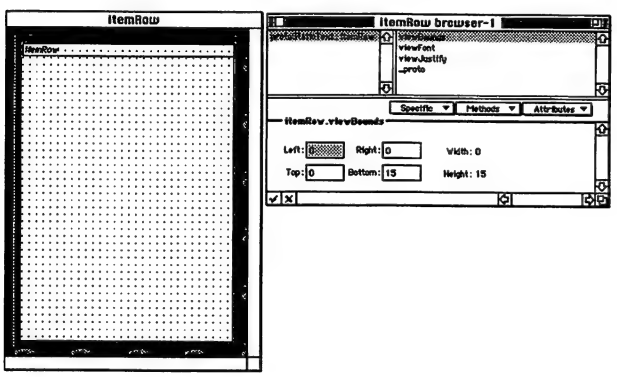


Figure 4.8 Creating a new user proto for items in the order template.

1. Create a user proto (see "Creating a User Proto" on page 361), save the file as "itemRow", and add it to your project.
2. Draw a `protoStaticText` in the layout window and name it "itemRow".
3. Create a browser, and delete itemRow's `text` slot (each template will need to provide a `text` slot).
4. Modify the itemRow's `viewBounds` slot to `{left: 0, right: 0, top: 0, bottom: 15}`.
5. Modify the `viewJustify` slot to horizontal: `parentRelativeFull` and `siblingNone`, vertical: `parentRelativeTop` and `siblingRelativeBottom`.
6. Modify the `viewFont` slot to `fancyFont12`. Your proto should now match the proto shown in Figure 4.8.

Using Item Proto

Now we have a proto that captures the commonality of `viewJustify`, `viewBounds`, and `viewFont` among items. Let's use the proto:

1. Delete the templates item1 and item2 from the order template in the detail layout file.
2. Draw two itemRow protos in the order template. Name them "item1" and "item2".
3. Add a `text` slot to item1 and give it the value `Coffee`.
4. Add a `text` slot to item2 and give it the value `Cake`.

That's all that needs to be done. In later chapters we'll add behaviors to the itemRow protos (such as adding code to handle taps). By having the behavior in the proto, we avoid duplicating code in each template.

Creating a Chair Proto

Next, let's create a proto for chairs. When you are done the new proto template should look like Figure 4.9.

1. Create a user proto, save the file as "chair", and add it to your project.
2. Draw a `clView` in the layout window and name it "chair".
3. Create a browser, and delete the `viewBounds` slot from the chair (each template will need to provide its own `viewBounds`).
4. Modify the `viewFormat` slot to pen: `1`, frame: `black`, fill: `white`.

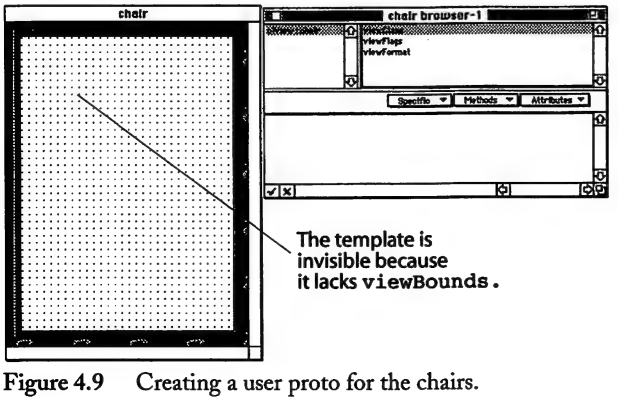


Figure 4.9 Creating a user proto for the chairs.

Using Chair Proto

Now we have a proto that captures the commonality of `viewFormat` among chairs. We will wait until a later chapter to add behavior to chairs; for now just use the new proto in the detail layout.

1. Edit the `viewClass` slot of each of the four chairs and change it to `chair`.

Building and Downloading WaiterHelper

If you build, download, and run your application, the table and chairs should look just as they did before. This would also be a good time to try out all the other features you have implemented in the detail view. (Don't forget to make the detail view visible and the overview invisible.)

Creating a Row Proto

In the overview template, each row is very similar. Let's create a user proto to capture that similarity. When you are through your new user proto should look like Figure 4.10.

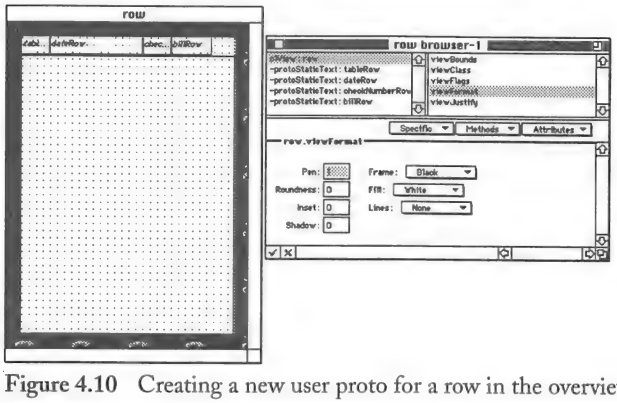


Figure 4.10 Creating a new user proto for a row in the overview.

1. Create a user proto, save the file as "row", and add it to your project.
2. Select the row1 template from the overview layout window, copy it, and paste it into the row proto template window. Rename the `clView` container "row".

3. Create a browser for row.
4. Modify the `viewJustify` slot to vertical: `parentRelativeTop` and `siblingRelativeBottom`. Your proto should now match the proto shown in Figure 4.10 (you may need to close the proto file and reopen it in order for NTK to display properly).

Using Row Proto

Now we have a proto that captures the commonality of `viewBounds`, `viewFormat`, `viewJustify`, and child templates among rows. Let's use the proto:

1. Delete the row1 and row2 templates from the rowContainer template.
2. Draw two (or more) row protos in the table template.
3. Again, there is no change visible on-screen from having used custom protos. We have saved space, while making maintenance and future enhancements easier.

Building and Downloading WaiterHelper

Now, build and download WaiterHelper again, this time to look at your newly revised overview. Remember to make the detail view invisible and the overview visible again.

Summary

The fundamentals of proto creation and use should now be clear to you. In this chapter you learned about both system and user protos and how they differ. You also learned about the numerous advantages of protos. Most importantly you learned that protos are the vehicles of code reuse on the Newton. They are also the best way to minimize memory use—still the Newton's most precious resource. Because protos are so important, they are worth constructing well. Good protos will be around for a long time and bad protos will be talked about for a long time. Wouldn't you prefer to have your name attached to the former, rather than the latter?

You also saw how we implemented some custom protos in WaiterHelper. We created new protos for pickers so that we could use full justification. We also made protos to use in both the detail and overview so that we could capture common template slots in one place. This reduces the size of our application and helps with program maintenance.

Chapter 5

The Fundamentals of NewtonScript

People should not travel until they have learned the language of the country they visit. Otherwise they voluntarily make themselves great babies—so helpless and so ridiculous.

—Ralph Waldo Emerson, paraphrased

NewtonScript, a new language designed specifically for the Newton, would be a delight on any platform. Its similarity to standard programming languages such as C and Pascal ensures an easy coding transition. At the same time, its innovative aspects—dynamic typing, frames, self, and so on—are constructs worth learning.

Indeed, we expect that once you are familiar with NewtonScript, you will be reluctant to return to many older languages.

NewtonScript owes its nature to its principal architect, Walter Smith. Lured from graduate school by the heady aroma of language creation and new hardware, Smith endowed NewtonScript with powerful and elegant features:

- It is object-oriented. The view system uses objects heavily.
- It has an unusual type of inheritance. Prototype inheritance is far more memory efficient than class-based inheritance. We'll cover inheritance in detail in Chapter 6.
- It is dynamic. You do not declare variable types—values have type, but variables do not. A variable can hold different types at different times, however.
- It is a full-featured programming language.
- It is portable and machine independent.
- It provides garbage collection. Memory deallocation is handled for you.

This chapter comprehensively covers NewtonScript (some of the more esoteric details are left to your further reading). The order in which we describe the language is somewhat innovative, however. Disdaining the classic textbook structure of building from the simple constructs to the complex types, we are covering the good parts first. After a one-page description of all the language's constructs, we immediately discuss the innovative and complex aspects of NewtonScript. Simpler constructs are covered afterwards. For example, we use variables long before we tell you that NewtonScript allows them and the rules for their use.

Note: We believe that this approach greatly benefits the programmer already familiar with one or two programming languages—you get to learn about the important variations and features in the language first (the areas that require more study and are, quite frankly, the most fun).

To those of you less familiar with programming languages, simply read through the chapter more than once—ignoring terms you don't understand until they are described. Anything undefined on the first reading should be more comprehensible upon subsequent readings.

A Brief Overview of NewtonScript

NewtonScript has a full range of features:

Frames	These are unordered structures and somewhat similar to Pascal records or C structs
Arrays	Ordered data structures, like arrays elsewhere
Built-in Functions	Standard string, real, array, frame, and integer functions
Symbols	Variable names—available at run time
Basic Types	Integers, booleans, and characters
Complex Types	Reals, arrays, frames, strings
Operators	All the standard ones
Conditional Statements	If then, if then else
Looping Statements	Loop, for, while, repeat, and some new ones

We will cover each of these in detail, starting with frames and then moving right on down the list.

Frames

```
{slotname: value, slotname2: value, …}
```

This data structure is the elastic frame that binds all NewtonScript programs together. Here is a basic definition of a frame:

- *A frame is a dynamic, unordered collection of named values.*
- Each element in a frame is called a slot and is composed of two things—a name and a value.

- A slot can hold any type of value, including another frame. Thus, frames can be (and often are) deeply nested structures.
- Frame syntax uses { }, frame elements are separated by commas, and the slot name is followed by a colon.

Frames are somewhat similar to a C struct or Pascal record. They are different in that they are truly dynamic—slots can be added or removed at any time.

Creating Frames

Creating a frame is so simple as to be disconcerting—`{ }` creates a frame. Thus:

```
x := {};
```

creates a frame and assigns it to `x`. Here is another example, this time a frame assigned to a variable `v`:

```
v := {left: 10, right: 20, top: 10, bottom: 100};
```

The frame has four slots, each containing an integer. This frame is more readable when formatted vertically:

```
v := {
   left: 10,
   right: 20,
   top: 10,
   bottom: 100,
};
```

Optional comma

Notice that the last slot has a trailing comma (`,`). When you format your frames with one slot per line, it is handy to end each line with a comma. Adding or deleting slots later is much easier when you do not have to treat the last line differently.

Frames within Frames

As we said, a slot can contain another frame. For instance:

```
v := {
   otherFrame: {
      x: "13",
      y: "20",
   },
   z: 5,
};
```

In this example, v has two slots, `otherFrame` and `z`. `otherFrame` is also a frame with two slots of its own, `x` and `y`.

A frame can also be referred to in more than one other frame:

```
sf:= {
   longitude: 37.48,
   latitude: 122.24,
};

la:= {
   longitude: 34.4,
   latitude: 118.15
};

earthquakes:= {
   recent: sf,
   realRecent: la
};

laSpec:= {
   city: "Los Angeles",
   location: la
};
```

Here you have two frames, `sf` and `la`, which are elements of another frame, `earthquakes`. `la` is simultaneously the value of a slot in `laSpec` as well. Figure 5.1 displays a graphic representation of the relationship between these four frames.

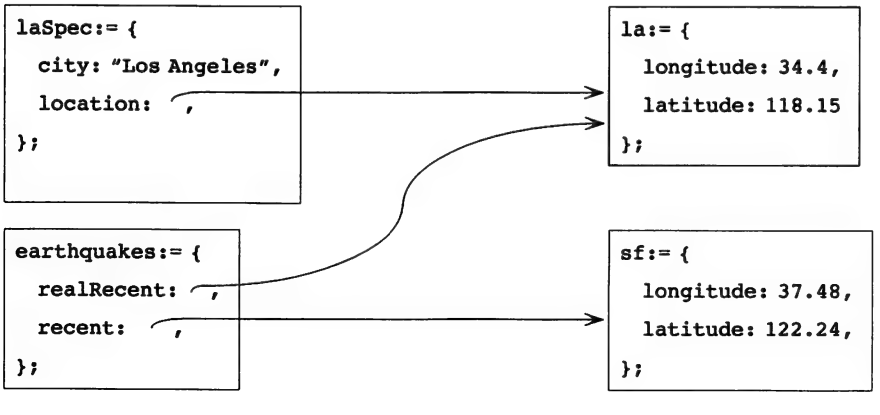


Figure 5.1 Frames pointing at other frames.

Accessing Slots

---

frame.slot

You use the dot (.) operator to access a slot (a frame element). For example:

```
x:= v.left + 3;
```

The code first retrieves the value of the `left` slot within v, adds 3to it, and then assigns it to x.

Of course, since any frame can appear to the left of the dot operator, you could even write this convoluted code:

```
{left: 10, right: 20, top: 10, bottom: 100}.left
```

in a Byzantine attempt to obtain the value `10`.

There are also a number of other ways to access slot values. They differ in how inheritance affects them and are covered in the discussion of inheritance in "Accessing Slots" on page 163.

Accessing a Nonexistent Slot

You will not get an error if you try to access a nonexistent slot. If the slot isn't in a frame, the return value is `NIL`. Consider a frame, v, that does not have an x slot:

```
v := {name: "Neil", height: 73.25, children: 2};
Print(v.foo);
```

Accessing a nonexistent slot results in the value `NIL`; `Print` prints `NIL`.

Because of this, you cannot use the dot operator to distinguish between nonexistent slots and existing slots whose values are `NIL`. To distinguish between the two, you must use the techniques discussed in "Slot Existence" on page 114.

Creating Slots

Creating a slot is as disconcertingly easy as creating a frame. Just use a new slot name in the frame in an assignment. To add the `foo` slot to our v frame, do this:

```
v := {name: "Neil", height: 73.25, children: 2};
v.foo := 17;
```

Creates a slot `foo`

`foo` now exists in v.

Note: It is slower to add a slot on the fly than it is to assign to an existing slot (memory allocation is necessary when the frame gets a new slot). The prudent course is obvious: if you know which slots you need, create them when you create the frame.

Multilevel Slot Creation

A subtle, but important aspect, of easy frame and slot creation becomes evident when you assign to a slot several levels deep. Here is the rule:

- *NewtonScript can create a whole hierarchy of frames or slots with just one assignment statement.*

Consider the following, slightly foolish example. In the first statement, you create a frame x with one slot, a. Next, you assign an integer value of 5 to a slot that is deeply nested within x:

Creates four frames

```
x := {a: 1};
x.b.c.d.e.f := 5;
```

Perhaps, a cascading result is not what you expected, but it is what you get. The second assignment statement actually creates four frames just so it can carry out the assignment. Figure 5.2 shows the frames in memory after the code executes.

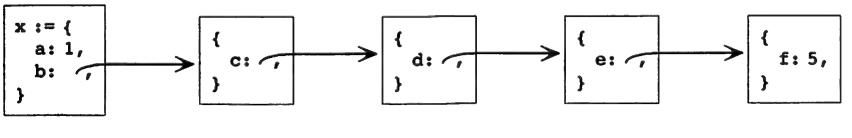


Figure 5.2 After executing `x.b.c.d.e.f := 5`.

When NewtonScript sees a multilevel slot access on the left hand side of an assignment statement, it creates those frames which don't exist. Each of these newly created frames has a single slot in it.

Removing Slots

```
RemoveSlot(frame, 'slotName);
```

To remove a slot, you use the `RemoveSlot` function. For example, consider the frame:

```
x := {a: 1, b: 2};
```

To remove the slot `a`, call `RemoveSlot` this way:

```
RemoveSlot(x, 'a);
```

Notice the single quotation (`'`) before the slot name; it is required. It specifies that `a` is a symbol and as such is not to be evaluated. For information on symbols, see "Symbols" on page 120.

Slot Existence

```
frame.slot exists
```

If you want to test for the existence of a slot in a frame, you use the `exists` operator. Given this frame:

```
x := {a: 1, b: 2};
```

you might use `exists` in this manner:

```
if x.a exists then
   Print("a is a slot in x");
```

evaluates to `true` since `a` is a slot in `x`

or:

```
doesFooExist := x.foo exists;
if doesFooExist then
   Print("uh-oh, foo shouldn't be a slot in x");
```

evaluates to `NIL` since `foo` is not a slot in `x`

You can also determine slot existence in other ways. The methods vary, however, in how they deal with inheritance (for a discussion of the alternatives see "Testing for the Existence of Slots" on page 163).

Frames in WaiterHelper

In light of what you know about frames, the structure of a NewtonScript application should now be easier to understand. *An application is a very large frame.* It is made up of a number of slots, many of which contain frames themselves. For example, look at a slice of the frame structure of the WaiterHelper application when it is open on the Newton (see Figure 5.3). Notice that `main` is a frame that has several slots, including two slots named `detail` and `overview`. Both `detail` and `overview` are themselves frames with slots of their own.

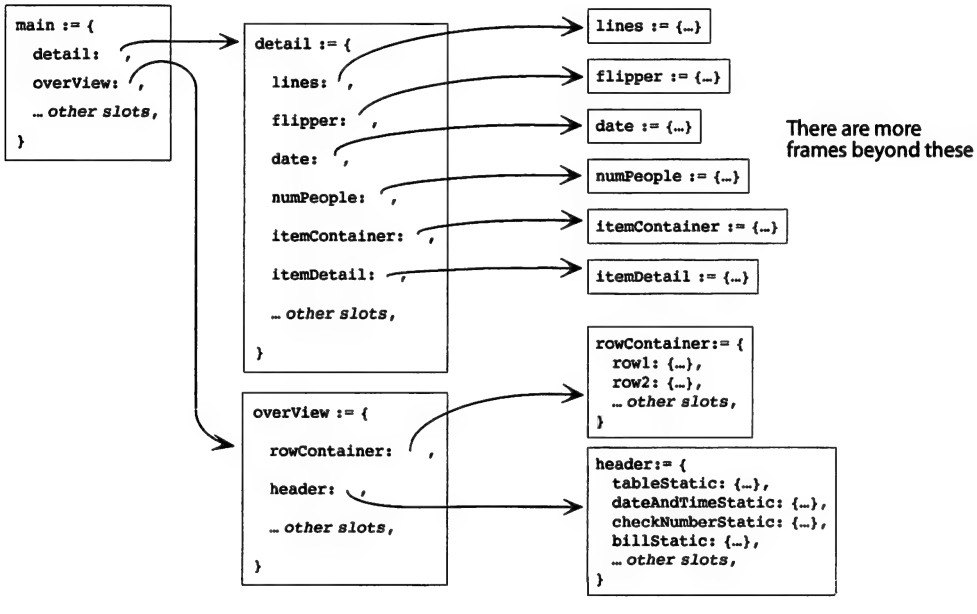


Figure 5.3 Some of the frames in WaiterHelper.

You will be adding slots to your application frames which hold variables, and slots containing methods that are needed by the application. Even the data that you store from within your application is stored in frames.

Their very pervasiveness marks frames as the core of a NewtonScript program.

Arrays

```
[element,element2, element3, element4, …]
```

An array in NewtonScript is an ordered collection of values. NewtonScript arrays are similar to C or Pascal arrays, but much more powerful. Their dynamic nature—arrays can grow or shrink—and their versatility—elements in an array need not be of the same type—makes them much more functional than their traditional counterparts. Here are three examples of arrays:

```
mixer:= ["Hello", 5, 4.32, "World!", NIL]
desserts:= ["cake", "pie", "flan", "ice cream",]
x : = [15, "xyz", {a: 2, b: 3}];
```

You can add, delete, modify, or retrieve elements from arrays.

You'll use arrays rather than frames if the order of elements is important.

Creating Arrays

Like frames, you create arrays just by using the array square brackets, `[ ]`. For instance, this creates a new empty array:

```
new := [];Accessing array elements is similar to
most languages:


y := x[0];
z := x[1];
zz := x[2];
```

As you might have guessed, NewtonScript array indexes start at 0 (the first element is at index 0). If you need to find the length of an array, you can use the built-in `Length` function. Here, is an example which prints out the elements of an array:

```
x := [15, "xyz", {a: 2, b: 3}];
for i := 0 to Length(x) - 1 do
   Print(x[i]);
```

Though this code works, there is a much better way to iterate through the elements of an array (see "Iterating with foreach" on page 122).

Adding Elements

You can increase the number of elements in an array by using one of two methods: `AddArraySlot` or `SetLength`.

AddArraySlot

`AddArraySlot(`*`array, value`*`)`

`AddArraySlot` appends a single element to the end of the array:

```
x := [15, "xyz", {a: 2, b: 3}];
AddArraySlot(x, 33);
```

After executing the last line, `x` has four elements; the last of which is `33`.

SetLength

`SetLength(`*`array, numberOfElements`*`)`

If you need to add a number of elements, you should use `SetLength`. This function adds a number of `NIL` elements simultaneously:

```
x := [15, "xyz", {a: 2, b: 3}];
SetLength(x, 10);
```

After executing `SetLength`, `x` has ten elements the last seven of which are `NIL`.

Note: If you are adding a number of elements to an array, it is quicker to use `SetLength` to specify the array's final size, and then assign each element with `[ ]`. This makes sense when you realize that each call to `AddArraySlot` requires new memory allocation, while using `SetLength` requires it only once.

RemovingElements

There are two functions you can use to remove elements: `ArrayRemoveCount` and the versatile `SetLength`.

ArrayRemoveCount

`ArrayRemoveCount(`*`array, beginningIndex, numElements`*`)`

The function `ArrayRemoveCount` removes one or more elements from anywhere within an array:

```
x := [15, "xyz", {a: 2, b: 3}];
ArrayRemoveCount(x, 0, 2);
```

After the code has executed, `x` has only one element: `x := [{a: 2, b: 3}]`.

SetLength

`SetLength(`*`array, numberOfElements`*`)`

In addition to adding elements, `SetLength` can remove them. If the *`numberOfElements`* parameter is less than the array's current length, the array is truncated to the *`numberOfElements`*.

```
x := [15, "xyz", {a: 2, b: 3}];
SetLength(x, 1);
```

After the code has executed, `x` has only one element: `15`. Note that `SetLength` can only remove elements from the end of the array.

Using Arrays as Sets

There is a handful of functions that allow the use of arrays as sets. A set is a collection of unique elements (no element is repeated). Here are some brief descriptions of these functions.

SetAdd

```
SetAdd(array, valueToAdd, uniqueOnly)
```

If *uniqueOnly* is NIL, this routine adds *valueToAdd* to *array*. If *uniqueOnly* is not NIL, the routine adds *valueToAdd* only if it is not already present in *array*.

SetRemove

```
SetRemove(array, valueToRemove)
```

This function removes *valueToRemove* from *array* if it is present.

SetUnion

```
SetUnion(array1, array2, uniqueOnly)
```

This function returns a new array that is the union of *array1* and *array2*. If *uniqueOnly* is not NIL, duplicates are eliminated.

SetDifference

```
SetDifference(array1, array2)
```

This function returns a new array with those elements in *array1* that are not in *array2*.

SetContains

```
SetContains(array, value)
```

Returns the index in *array* if *value* is present in *array*, NIL otherwise. The function uses equality testing (=) to determine whether *value* is in *array*.

SetOverlaps

```
SetOverlaps(array1, array2)
```

If any element of *array1* is present in *array2*, this function returns the first index. Otherwise, it returns `NIL`.

Symbols and Path Expressions

Symbols

```
'symbolname
```

You specify that an identifier is a symbol by preceding it with a single quote (`'`). The quote suppresses the evaluation of the identifier (no variable lookup occurs). One use of symbols is as a way to provide distinct values. For instance:

```
x := 'red;
...
if x = 'red then
    ...
else if x = 'blue then
    ...
else if x = 'green then
    ...
```

To convert a string to a symbol, use the function `Intern`, to convert a symbol to a string, use the function `SPrintObject`. For instance:

```
Print(Intern("red"))
red

Print(SPrintObject('red))
"red"
```

Path Expressions

---

frame.(*pathExpression*)

You use path expressions to refer indirectly to slots. A path expression can be any one of these three:

- an integer. For example, `3`.
- a symbol. For example, `'x`.
- an array of class pathExpr containing symbols and/or integers.

An example of the last type of path expression would be:

```
[pathExpr: 'x, 'y, 'z]
```

It is an acceptable shortcut to separate symbols by periods (`.`) when the array doesn't contain any integers. Thus, the following is equivalent to the previous array:

```
'x.z.y
```

You will use this type of indirection when you want to decide what slot to access at run time instead of compile time. Here is an example that uses this type of expression:

```
func()
begin
   local aFrame := {x: 1, y: 2, z: 3};
   local slotToAccess;
   if … then
      slotToAccess := 'x;
   else if … then
      slotToAccess := 'y;
   else
      slotToAccess := 'z;
   …
   aFrame.(slotToAccess) := 6;
end;
```

Assigns to `x`, `y`, or `z` depending on the value of `slotToAccess`

Using Path Expressions for Constants

Path expressions are most commonly used as a way to specify a slot in a constant. For instance:

```
func()
begin
   constant kSlotToUse := 'theSlot;

   frame.(kSlotToUse) := 5;
   ...
   total := total + frame.(kSlotToUse)
   ...
end;
```

This use of path expressions is especially nice if you have an application-wide constant that is used in many different methods.

Iterating with foreach

Have you ever incorrectly written the bounds of a `for` loop? With NewtonScript, this is a relic of the past. As we said earlier, you do not use for loops to iterate the elements in an array or a frame. Instead, you use the unique NewtonScript construct, `foreach`.

foreach with Arrays

```
foreach element in array do statement
foreach element in array collect statement
```

When using traditional `for` loops to iterate over an array, it is easy to get the beginning or ending index wrong. Consider the irksome fact that only one of the following `for` loops correctly indexes through `array` elements:

```
for i := 1 to Length(array) do
   Print(array[i]);

for i := 1 to Length(array) - 1 do
   Print(array[i]);

for i := 0 to Length(array) - 1 do
   Print(array[i]);
```

Correct

```
for i := 0 to Length(array) do
   Print(array[i]);

for i := 1 to Length(array) do
   Print(array[i - 1]);
```

To avoid this morass, NewtonScript provides you with a bullet-proof `for` loop—`foreach`. A `foreach` loop iterates over each element in an array automatically. Here is the same example array using the much simpler syntax of `foreach`:

```
foreach element in array do
   Print(element);
```

The `element` variable is a loop variable similar to the `i` loop variable found in the previous examples. There is a crucial difference between the two, however. Element does not take on successive *index values* (`0`, `1`, etc.), but rather takes on successive *element values* (`array[0]`, `array[1]`, etc.).

As is the case with a standard `for` loop, the loop variable in `foreach` can be named anything—be it a single letter or a song:

```
foreach yellowSubmarine in array do
   Print(yellowSubmarine);
```

foreach with Frames

```
foreach value in frame do statement
foreach value in frame collect statement
```

Thankfully, you can use the same foreach loop to iterate over slots in a frame. The following example prints all the slots in the frame `x`:

```
x := {a: 3, b: 9, c: 15};
foreach value in x do
   Print(value);
```

It prints:

```
3
9
15
```

The loop variable takes on each of the slot values in the frame. Note that you cannot count on the order in which the elements are accessed—it is undefined.

foreach with Slot Names

```
foreach slotname,value in frame do statement

foreach slotname,value in frame collect statement
```

In some cases you may want to iterate over each slot's name (actually, the symbol) as well as its value. An alternate form of the `foreach` loop provides this capability. This version takes two loop variables rather than one. For each iteration, the first variable takes on the slotname (the symbol of the slot), while the second takes on the slot value. Here is a frame:

```
x := {a: 3, b: a, c: 15);
```

with this variation of `foreach`:

```
foreach slotname,value in x do
   Print(slotname && value);
```

This will print out:

```
"a 3"
"b 9"
"c 15"
```

Here is another example of using `foreach`. It is a rather slow way of copying the slots from one frame to another. First, here are the frames:

```
x := {a: 3, b: 9, c: 15};
y := {};
```

Now, using `foreach`, here is the actual copying:

```
foreach symbol, frameValue in x do
   y.(symbol) := frameValue;
```

This `foreach` with two loop variables also works with arrays; the first variable takes on the index number while the second takes on the value at that index. For instance, here is a way to pretty-print an array:

```
x := [3, 9, 15, "c"];
foreach indexValue, e in x do begin
   Write(indexValue);
   Write(": ");
   Print(e);
end;
```

When you execute this, it prints:

```
0: 3
1: 9
2: 15
3: "c"
```

There are also versions of foreach that work up the inheritance chain. For information on them, see "The deeply Version of foreach" on page 158.

Types

Values in NewtonScript are stored as 32-bit references (2 of those bits are used for meta information). There are two types of references:

Immediate	Those values that can completely fit in 30 bits: integers, characters, and booleans.
Nonimmediate	Those values that are too big for 30 bits: strings, symbols, real numbers, frames, arrays, and binary objects (like bitmaps or functions).

When you copy a value with an assignment statement, the 32-bit reference is copied. If the value fits in the reference (immediate), then the value is copied. If the value is larger than the 32-bit reference (nonimmediate), then only a pointer to the value is copied. Likewise, when you pass a parameter, the 32-bit reference is passed. Consider the following assignments:

```
a := [1, 2, 3];
b := a;
```

The values of a and b are the same 32-bit reference—they are both pointing at the same array (see Figure 5.4).

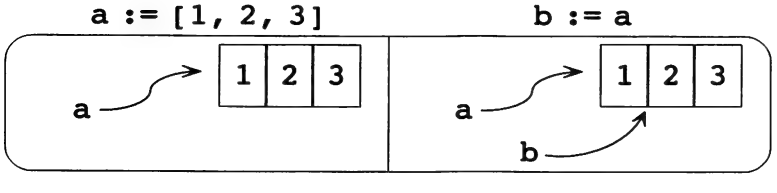


Figure 5.4 Two variables pointing at the same array.

If you were to change one of the elements using the a array, the array element is also changed for b:

```
a[0] := 5;
Print(b);
[5, 2, 3]
```

What Types Can Be Modified

You can modify the contents of strings, frames, arrays, and binary objects, but not the contents of symbols, real numbers and immediates. For example, if you pass an array as a parameter to a function, when the function returns, the array could be modified:

```
ModifyAnArray := func(anArray)
begin
   anArray[0] := 5;
end;
```

On the other hand, if you pass a real number as a parameter to a function, when the function returns, the real number cannot have changed. Here is the reason:

- *There are operators and functions that can modify the contents of a string, frame, array, and binary object. No such operators or functions exist for symbols, real numbers, or immediates.*

In Chapter 7, we'll also see that attempts to modify the contents of an object in ROM is a common error while learning Newton Programming

Clone/DeepClone

`Clone(value)`
`DeepClone(value)`

The `Clone` function returns a one-level-deep copy of a reference. `DeepClone`, as its name implies, returns a multilevel recursive copy of a reference. When you have an immediate reference, `Clone` and `DeepClone` have the same effect—they just return their argument.

For a nonimmediate reference, `Clone` returns a copy of the memory block to which the reference points. `DeepClone` returns the same copy and recursively

copies any references within that block. Thus, for arrays and frames, `Clone` and `DeepClone` operate differently as these types store references within them. Figure 5.5 shows the difference between `Clone` and `DeepClone` and in contrast to standard assignment.

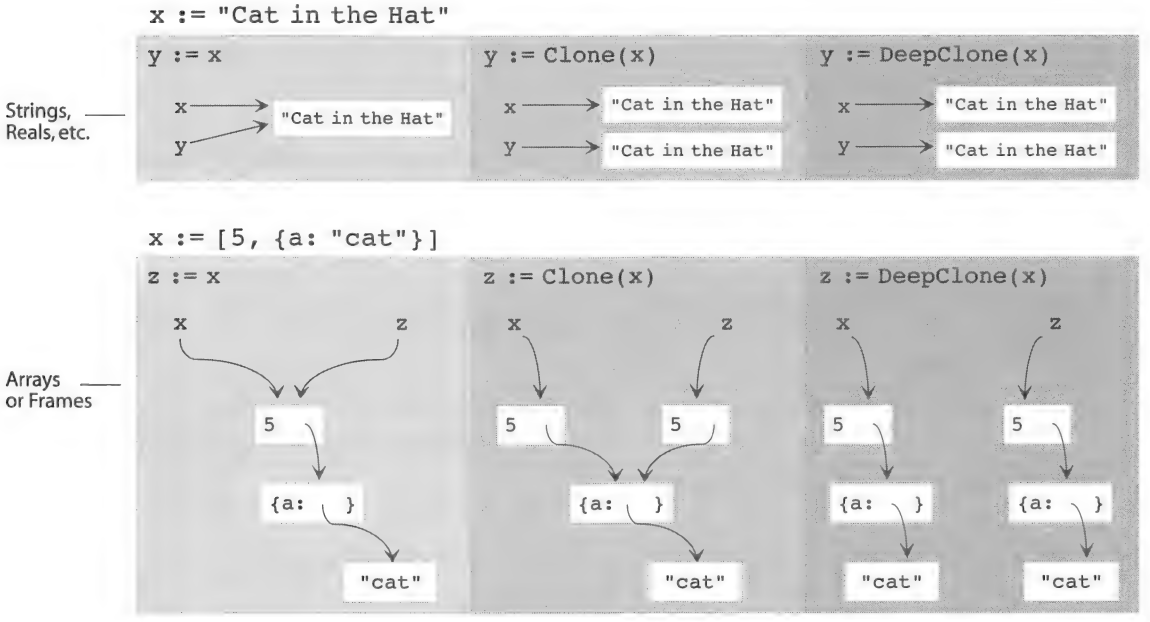


Figure 5.5 The differences between assignment, `Clone`, and `DeepClone`.

Methods

```
func(parameterFirst, …, parameterLast)
begin
    code
end
```

A *method* is simply a slot within a frame that contains a function. Here is an example of a method, `Max`, that returns the larger of two values:

```
aFrame := {
   Max: func(a, b)
   begin
      if b > a then return b;
      else return a;
   end,
};
```

A function always returns a value. If an executing function has a return statement, then that is the function's value. If the executing function does not have a formal return statement, then the last executing statement is the function's value. A couple of examples should make this clear. You could rewrite the above `Max` function in this way:

```
func(a, b)
begin
   if b > a then b;
   else a;
end;
```

The returns are missing

The `return` statements can be removed since the value of the `if` statement is either the value of `b` or the value of `a` (see "if/then/else Statements" on page 136). You can produce an even more petite `Max`:

```
func(a, b)
   if b > a then b
   else a;
```

With one-statement functions, the `begin` and `end` are unnecessary. Since the if/else is now only one statement, you can snip the `begin` and `end`.

While this last form of `Max` is the smallest, it is not necessarily the best. The original `Max` is easier to read for many people.

Local Variables

Any of your functions can have local variables using the `local` syntax:

```
func(a, b, c)
begin
   local maxSoFar := a;

   if b > maxSoFar then maxSoFar := b;
   if c > maxSoFar then maxSoFar := c;
   return maxSoFar;
end;
```

It is not necessary to assign a value to a local variable at the time you declare it. All locals are initialized to `NIL` at the beginning of a function.

The scope of a local variable is the entire function. Though this may seem weird, you can use variables before they are declared:

```
func(a, b, c)
begin
   maxSoFar := a;

   if b > maxSoFar then maxSoFar := b;
   if c > maxSoFar then maxSoFar := c;
   return maxSoFar;
   local maxSoFar;
end;
```

Good style dictates that you declare your local variables before you use them; otherwise readers of your code will be quite confused.

Message Sending

`frame:MethodName(parameters)`

To call a method, you send a message to the frame containing the method. Sending a message is straightforward:

```
result := aFrame:Max(a, b);
```

The above code sends the `Max` message to `aFrame`.

Sending a message requires a frame (sometimes called an object) and the name of a function, along with parameters, if any. Between the frame and the function name is a colon (`:`).

The distinction between a method and a message is simple: a method is the actual function, while a message is a call of that function. As you'll see in Chapter 6, it is quite common to have many methods with the same name. When you send a message, the method that actually executes depends on inheritance. For example, you may have a `Display` method in many of your application template frames. Just as you would expect, the Display method that executes depends on which frame you send the message to and the rules of inheritance.

Note: A common mistake NewtonScript programmers make is to type a . instead of a : when sending a message. Here is the chatty error you get:

```
//-- Error:  "<input>", Line 1, syntax error--read '(', but wanted end-of-
file, '&', ')', '*', '+', ',', '-', '.', '/', ':', ';', '<', '>', '[',
']', '}', SYMBOL, END, THEN, ELSE, ONEXCEPTION, TO, BY, UNTIL, DO, WITH,
ASSIGN, AND, OR, LEQ, GEQ, EQL, NEQ, EXISTS, AMPERAMPER, DIV, MOD, LS
```

Just remember when you see this error to check for periods instead of colons in your messages.

self

self is an important part of NewtonScript. It is a special variable that is available to a method while it is executing. Its value is always the frame to which the message was sent. Because of self, methods can have access to the frame that owns them. Look for example at the frame, `bankBalance`, and its three methods, `Deposit`, `Withdraw`, and `Balance`:

```
bankBalance := {
   total: 0,
   Deposit:     func(amount)
                begin
                   self.total := self.total +
amount;
                end,
   Withdraw:    func(amount)
                begin
                   self.total := self.total -
amount;
                end,
   Balance:     func()
                   return self.total,
};
```

To send messages to these various methods requires no more than:

```
bankBalance:Deposit(50);
bankBalance:Withdraw(20);
bankBalance:Balance();
30
```

As each of the three methods executes, `self` is the `bankBalance` frame; it is the frame to which each message is sent. Thus, `self.total` is the `total` in the `bankBalance` frame.

Variable Lookup Rules

You will not be littering your methods with numerous references to `self`, however (the previous example is for instructional purposes only). You will be relying upon NewtonScript's lookup rules to determine which variable is being accessed and what its value is. Here are the lookup rules for finding a variable:

1. Local variables or parameters are searched first.
2. If no local variable is found, then global variables are searched.
3. If no global variable is found, slot names are searched using inheritance rules (covered in Chapter 6). The lookup starts in the frame that is `self`.

If the variable still isn't found, a run-time error is generated.

The rules are slightly different when you are not searching for a variable but rather are *assigning to one*. The first three rules remain the same, but a fourth, very important rule has been added:

1. Local variables or parameters are again searched first.
2. Global variables are again searched second.
3. Slot names are searched using inheritance rules. The lookup still starts with `self`.
4. If no variable has been found, *a local variable is created with that name.*

Effects of Variable Lookup Rules

These lookup rules have a number of repercussions.

- *Methods need not use* `self` *to access or set slots within their own frame.*

You can now rewrite the bankBalance frame and its three methods to take advantage of these rules:

```
bankBalance := {
   total: 0,
   Deposit:     func(amount)
                begin
                   total := total + amount;
                end,
   Withdraw:    func(amount)
                begin
                   total := total - amount;
                end,
   Balance:     func()
                   return total,
};
```

Caution: See "Using self in a Method" on page 164 for a complete description on when to use `self` and when not to use it.

Another effect of the lookup rules involves variable declaration:

- *You are not currently required to declare local variables.*

Thus, you could rewrite the `Max` method without ever declaring `maxSoFar`:

```
aFrame := {
   Max3 :    func(a, b, c)
             begin
                maxSoFar := a;

                if b > maxSoFar then maxSoFar := b;
                if c > maxSoFar then maxSoFar := c;
                return maxSoFar;
             end,
}
```

The first time `maxSoFar` is assigned, the assignment lookup rules come into play. First, `maxSoFar` is looked for as a local, then as a global, then a slot in `aFrame`. When these three rules fail to produce `maxSoFar`, it is created as a local in `Max3`.

Nevertheless, you should still declare your local variables. There are a number of compelling reasons for this:

- It is harder to read and understand code that doesn't explicitly declare variables.

- If your frame has a slot with the same name as the variable, lookup will find the slot first—obviously not what you want.
- The initial assignment is slower. The entire set of lookup rules had to be exercised, instead of just using the first rule.
- Explicitly declared locals are optimized by the NewtonScript compiler—they can be accessed and assigned much quicker than undeclared locals created at run-time.
- Your code won't break if and when NewtonScript requires locals to be declared.

Additional NewtonScript Features

There are some additional features of NewtonScript you need to understand:

- Other standard types
- Strings
- Operators
- If /then and If /then /else statements
- Loops

Other Standard Types

Beyond frames, arrays, and symbols, NewtonScript also provides some other essential types. These are integers, reals, characters, and booleans. Here is a brief description of each:

Integer	A 30-bit integer that can hold values from -2^{29} to $2^{29}-1$.
Real	A 64-bit floating point number.
Boolean	There are two boolean constants, `true` and `NIL`. In boolean expressions, any non-`NIL` value is considered `true`. *Note: the integer* `0` *is not the same as the boolean* `NIL`.

Character	A character. Character constants begin with a `$` (e.g., `$a`, `$b`, `$c`). These are 16-bit characters that use the Unicode encoding scheme.

Strings, another NewtonScript type, require some additional discussion.

Strings

```
"Here is a string"
```

A string is a sequence of characters. There are also two string concatenation operators—`&` and `&&`. They both concatenate two strings; the `&&` operator inserts a space character between the strings.

Examples

Let us look at some examples of how you use NewtonScript strings:

```
Print("abc" & "def");
"abcdef"                    Inspector output
Print("abc" && "def");
"abc def"
```

The concatenation operators will convert their arguments into strings, if necessary. For example:

```
Print("abc" & 123);
"abc123"
Print("5 + 3 =" && 5 + 3);
"5 + 3 = 15"
```

As you would expect, there are a number of functions that operate on strings (for a complete list, see Appendix C on page 315). One of the most important of these is `StrLen`—a function that calculates the length of a string. For example:

```
Print(StrLen("abc"));
3
Print(StrLen(""));
0
```

A common error is to attempt to use `Length` rather than `StrLen` to determine the length of a string. Strings are stored as a sequence of two-byte charac-

ters, followed by two zero bytes. Thus, `Length` returns two plus twice the number of characters in a string.

For example:

```
Print(StrLen("abc"));
3
Print(Length("abc"));
8
Print(Length(""));
2
```

Length gives the wrong value.

Accessing String Elements

You access individual characters in a string using array syntax. The first character is located at index `0`:

```
myString := "abcde";
Print(myString[1]);
$b
myString[1] := $x;
Print(myString);
"axcde"
```

Although you can use array syntax with strings, strings are not arrays. Most mournfully, the foreach loop does not work. You will need to revert to an old-style `for` loop to iterate through the elements of a string:

```
for i := 0 to StrLen(s) - 1 do begin
   // do something to s[i]
end
```

Unicode

Newton is an international machine, so it uses the Unicode character encoding. In this international character-encoding standard, each character is represented with two bytes. The two-byte values from 1 to 127 represent standard ASCII values. For example, the letter 'A' in Unicode is represented with a high byte of 0 and a low byte of the ASCII value of A.

Beyond the 127 horizon, you will need to specify your characters using the Unicode designation:

$\uXXXX	where "XXXX" is the four character hexadecimal value that specifies a particular Unicode character. The $\u must preface the character.
"\uXXXX"	Within a string, \u enters Unicode mode. The four hexadecimal characters specify a particular Unicode character. \u exits Unicode mode, as well.

For example, to specify the seven-character string "`abc…def`" (containing an embedded '…' character, use:

```
"abc\u2026\udef"
```

Unicode for "…"

Operators

NewtonScript provides the standard arithme
integer divide (`div`), and integer remainder (`mod`). These and the rest of the NewtonScript operators are described in Table 5.1.

if/then/else Statements

```
if expression then statement
if expression then statement1 else statement2
```

NewtonScript's `if` statements are similar to those found in other languages. Here are some examples:

```
if a < b then
   min := a;
else
   min := b;

if a < b then
   x := y;

min := if a < b then
       a;
   else
       b;
```

Operator	*Operation*	*Associativity (if not left-to-right)*	*Example*
`.`	slot access		`myFrame.slot`
`:`	message send		`aView:Open()`
`:?`	conditional message send		`aView:?GetSize()`
`[]`	array de-reference		`myArray[5]`
`-`	negate		`-6`
`>>`	right-shift		`8 >> 2`
`<<`	left-shift		`3 << 1`
`*`	multiply		`5 * 6`
`/`	real division		`10 / 3.5`
`div`	integer division		`17735 div 6`
`mod`	integer remainder		`17735 mod 6`
`+`	add		`8 + 10`
`-`	subtract		`9.5 - 6.3`
`&`	concatenate string representations		`"6*3 = " & 6*3`
`&&`	same as `&`, but with a space		`"6*3:" && 6*3`
`exists`	variable and slot existence	none	`aFrame.foo exists`
`<`	less than		`5 < 10.3`
`<=`	less than or equal to		`5 <= 10.3`
`>`	greater than		`5 > 10.3`
`>=`	greater than or equal to		`5 >= 10.3`
`=`	equal to		`a = b`
`<>`	not equal to		`a <> b`
`not`	boolean not		`not a < 3`
`and`	boolean and (short-circuit)		`x and x.y <= 6`
`or`	boolean or (short-circuit)		`a > 2 or b < 6`
`:=`	assignment	right-to-left	`a := b`

Table 5.1 NewtonScript operators grouped in precedence order.

An `if` statement has a value—like all NewtonScript statements. Where an `else` is lacking, the value of the `if` statement is either:

- the value of the statement associated with the `if`, or
- `NIL`

If there is an `else`, then the value of the `if` statement is either:

- the value of the statement associated with the `if`, or
- the value of the statement associated with the `else`.

In the case where `else` matches the value, it is to the closest preceding `else`. For instance,

```
if a < b then if b < c then x := y else y := x;
```

is parsed as

```
if a < b then
   if b < c then
      x := y
   else
      y := x
```

The `else` belongs to the second `if`

rather than as

```
if a < b then
   if b < c then
      x := y
else
   y := x
```

not correct

There is one other important point about `if` statements:

- There is an optional semicolon before the `else`.

This optional syntax makes migration from other languages easier—some languages prohibit a semicolon at this point (like Pascal), while others require it (like C). The preferred approach is not to use the optional semicolon, however.

Loops

NewtonScript provides a number of different loop constructs. You've already seen the `foreach` loop. The other loop constructs are: `while`, `repeat`, `loop`, `for`, and a collect version of `for`.

Break

```
break [expression]
```

All loops can be terminated with a `break` statement. The break statement takes an optional expression that then becomes the value of the loop. Without a break statement, a loop has a `NIL` value. For instance, the following function returns `true` if its parameter (array or frame) contains a `NIL` slot:

```
HasNilSlot:= func(arrayOrFrame)
begin
   return foreach value in arrayOrFrame do
      if value = NIL then
         break true;
end;
```

while

```
while boolean-expression do looping-statement
```

The NewtonScript `while` loop is very similar to `while` loops in other languages:

```
e := q:Entry();
while e do
   e := q:Next();
```

The while loop evaluates its boolean expression at the top of the loop. If the expression is non-`NIL`, it evaluates the looping statement and then returns to the top of the loop. If the expression is `NIL`, it skips to the next statement. The looping statement may be executed zero or more times.

repeat

```
repeat
    looping-statement
until boolean-expression
```

The NewtonScript repeat loop is very similar to repeat loops in other languages:

```
e := q:Entry();
repeat
   :Foo();
   e := q:Next();
until e = NIL;
```

The repeat loop evaluates its boolean expression at the bottom of the loop. If the expression is `NIL`, it jumps back up to the repeat. If the expression is non-`NIL`, it skips to the next statement. The looping statement is always executed at least once.

loop

```
loop
    looping-statement
```

A `loop` repeatedly evaluates its looping statement until a `break` or `return` statement is executed. For example:

```
loop
   if Time() > kTimeToLeave then
      break;
```

for

```
for loop-variable := start to end do
    looping-statement

for loop-variable := start to end by index do
    looping-statement
```

The `for` loop initializes loop-variable to *`start`*. Each time through the loop, the loop variable is compared to *end*. If the loop variable is greater than *end*, the loop terminates. (If *`index`* is negative, it terminates when less than *end*.) Otherwise, the looping statement is executed and the loop variable is incremented by 1 (if *`index`* exists, the variable hops by *`index`*). Then, execution jumps back to the top, with a comparison to *end*.

The `start` and `end` expressions are evaluated exactly once. The looping statement is executed zero or more times. The loop variable is automatically created as a local variable for you (unless it is already local).

After the loop, the loop variable's value is undefined—don't rely on it.

for/foreach collect

```
foreach … collect
    looping-statement
```

The foreach statements have a `collect` version as well as the standard `do` version. The collect version, however, has a much different statement value. Each time through the loop, the value of the looping statement is collected into an array. The value of the `foreach` loop is then that array. The first array entry is the value of the looping statement the first time the loop executes, the second array entry is the value the second time through the loop, and so on. For example:

```
x := [1, 2, 3, 5];
y := foreach value in x collect
      value * 2;

Print(y);
[2, 4, 6, 10]
```

The Benefits of NewtonScript

At this point, you may be getting a feel for what a flexible and elegant language you have in NewtonScript. There are two further advantages that will make your life as a Newton programmer even easier:

- *NewtonScript is a portable language with built-in garbage collection.*

You will no longer create a new version of an application sorely for the sake of keeping abreast of new hardware. There are also no more memory-leak debugging nightmares.

Portability

NTK compiles NewtonScript into machine code for a stack-based, virtual machine. Each Newton contains an interpreter for this virtual machine. There are a couple of advantages to this approach:

- Applications you write today will run on Newtons of the future—even if they have different processors.
- Your compiled application is smaller than it would be if it were compiled for the native processor (the stack-based code is very compact).

There is one main disadvantage, however: your application is slower. There are a couple of different ways that this disadvantage can be addressed:

1. *The interpreter for this virtual machine can be optimized.*
   For example, some Smalltalk environments (which use the same virtual machine approach) convert the virtual machine code for a function to native code. The first time the function executes, the conversion is cached in native code.

2. *NTK could generate native code for the key, time-intensive functions in your application.*
   Your application would grow slightly larger, but would execute much quicker. It would continue to be portable, since these key functions would be compiled in both native code and virtual machine code versions.

We suspect that the future holds some type of optimization for the Newton. These are the two most likely approaches.

Garbage Collection

You never explicitly deallocate memory in NewtonScript. Instead, memory that is no longer referenced is available for reallocation.

Memory is allocated when you execute code that creates a non-immediate (e.g., frame, array, or function). For instance, this code creates memory for two data structures (one array and one frame):

```
func()
begin
   local x := {j: 3, k: [1, 2, 3], l: 5};
end;
```

In addition, a number of functions allocate memory. For example, `Clone` and `DeepClone` both allocate memory.

Consider the following code, which demonstrates garbage collection:

Both `y` and `x.c` point to the same array

Only `y` points to the array

The array is no longer referenced. As `x` and `y` are no longer present, the frame is not referenced either.

```
func()
begin
   local x := {a: 2, b: 3, c:[1, 2, 3]};
   local y := x.c;

   x.c := nil;
   y := 5;
   return;
end;
```

Garbage collection works by starting with all the accessible variables. It then follows those variables and any frames or arrays until it can go no further. This determines all memory objects that can be reached; any others are unreachable and are marked for reuse.

Garbage collection occurs on a periodic basis as memory is needed. You don't need to do anything; the process is automatic. There is a `GC()` call that initiates garbage collection; it is improbable that you will ever need to use it, however.

Writing Code for WaiterHelper

WaiterHelper must keep track of a number of different menu items, their prices, names, and categories. We can write an object that stores this and provides an object-oriented interface to this menu information. Since all the menu information will be encapsulated in this object, changes to the menu can be made in one place.

You are going to create this new code in the Inspector, rather than putting it directly into WaiterHelper. This will give you an opportunity to become familiar with the Inspector and all its vices and virtues.

1. Create this very large frame in the Inspector:

```
menu := {
   // converts an item name to an item symbol
   ItemToItemSymbol:
      func(itemName)
         foreach itemSymbol, item in items do
            if StrEqual(itemName, item.string) then
               return itemSymbol,

   // converts an item symbol to an item name
   ItemSymbolToItem:
      func(symbol)
         return items.(symbol).string,

   // returns the category symbol associated with an item symbol
   ItemSymbolToCategorySymbol:
      func(symbol)
         return items.(symbol).category,

   // returns the category name associated with an item symbol
   ItemSymbolToCategory:
      func(symbol)
         return :CategorySymbolToCategory(
             items.(symbol).category),

   // returns the price of an item symbol
   ItemSymbolToPrice:
      func(symbol)
         return items.(symbol).price,

   // converts a category name to a category symbol
   CategoryToCategorySymbol:
      func(category)
         return Intern(category),

   // converts a category symbol to a category name
   CategorySymbolToCategory:
      func(categorySymbol)
         return categories.(categorySymbol),

   // returns an array of item names associated with a category
   CategoryToItems:
      func(category)
      begin
         local categorySymbol :=
             :CategoryToCategorySymbol(category);
         local itemArray := [];
         foreach item in items do
            if item.category = categorySymbol then
               AddArraySlot(itemArray, item.string);
         return itemArray;
      end,
```

```
// returns an array of all category names
GetCategories:
   func()
   begin
      return foreach name in categories collect
          name;
   end,


categories: {
   none:       "None",
   appetizer:  "Appetizer",
   soup:       "Soup",
   entree:     "Entree",
   salad:      "Salad",
   side:       "Side",
   dessert:     "Dessert",
   beverage:   "Beverage",
},

items: {
  none: {
      string: "None", category: 'none, price: 0.00},

  friedZuchinni: {
     string: "Fried Zucchini", category: 'appetizer, price: 1.50},
  potatoSkins: {
    string: "Potato Skins",   category: 'appetizer, price: 2.50},
  artichokes: {
    string: "Artichokes",     category: 'appetizer, price: 3.50},
  roastedPeppers: {
     string: "Roasted Peppers",category: 'appetizer, price: 1.75},
  quesadilla: {
    string: "Quesadilla",     category: 'appetizer, price: 2.50},

  soupDuJour: {
    string: "Soup du Jour",        category: 'soup, price: 1.67},
  tomatoWhiteBean: {
    string: "Tomato & White Bean", category: 'soup, price: 1.67},
  potatoCornChowder: {
    string: "PotatoCornChowder",   category: 'soup, price: 1.67},
  gazpacho: {
     string: "Gazpacho",           category: 'soup, price: 1.67},
  blackBean: {
    string: "Black Bean",          category: 'soup, price: 1.67},
  minestrone: {
    string: "Minestrone",          category: 'soup, price: 1.67},
  leekPotato: {
    string: "Leek Potato",         category: 'soup, price: 1.67},

  spinachLasagna: {
    string: "Spinach Lasagna",   category: 'entree, price: 1.50},
  garlicRavioli: {
    string: "Garlic Ravioli",    category: 'entree, price: 1.50},
  greekPizza: {
     string: "Greek Pizza",       category: 'entree, price: 1.50},
  grilledSwordfish: {
     string: "Grilled Swordfish", category: 'entree, price: 1.50},
```

```
spicySeaBass: {
   string: "Spicy Sea Bass",     category: 'entree, price: 1.50},
tunaMarseilles: {
   string: "Tuna Marseilles",    category: 'entree, price: 1.50},
ratatouille: {
   string: "Ratatouille",        category: 'entree, price: 1.50},
vegetablePie: {
   string: "Vegetable Pie",      category: 'entree, price: 4.95},

grilledPotatoFeta: {
  string: "Grilled Potato w/ Feta", category: 'salad, price: 4.95},
lemonArtichoke: {
  string: "Lemon Artichoke",        category: 'salad, price: 4.95},
broccoliTomato: {
  string: "Broccoli & Tomato",      category: 'salad, price: 4.95},
chineseNoodle: {
  string: "Chinese Noodle",         category: 'salad, price: 4.95},
stuffedTomato: {
  string: "Stuffed Tomato",         category: 'salad, price: 4.95},
chickPeaChilies: {
  string: "Chick Pea w/ Chilies",   category: 'salad, price: 4.95},

threeBean: {
   string: "Three Bean",        category: 'salad, price: 1.50},
spicyFries: {
   string: "Spicy Fries",       category: 'side, price: 1.50},
wildRice: {
   string: "Wild Rice",         category: 'side, price: 1.50},
roastedPotatoes: {
   string: "Roasted Potatoes", category: 'side, price: 1.50},
steamedGreens: {
   string: "Steamed Greens",    category: 'side, price: 1.50},
winterSquash: {
   string: "Winter Squash",     category: 'side, price: 1.50},
onionRings: {
   string: "Onion Rings",       category: 'side, price: 1.50},
mashedPotatoes: {
   string: "Mashed Potatoes",   category: 'side, price: 1.50},

pie: {
   string: "Pie",                      category: 'dessert, price: 1.50},
pieIceCream: {
  string: "Pie w/ Ice Cream",          category: 'dessert,price: 1.50},
tangerineSorbet: {
  string: "Tangerine Sorbet",          category: 'dessert,price: 1.50},
flanCaramelSauce: {
  string: "Flan w/ Caramel Sauce",     category: 'dessert,price: 1.50},
persimmonPudding: {
  string: "Persimmon Pudding",         category: 'dessert,price: 1.50},
raspberriesFigs: {
  string: "Raspberries & Figs",        category: 'dessert,price: 1.50},
lemonPotsDeCreme: {
  string: "Lemon Pots de Creme",       category: 'dessert,price: 1.50},
apricotCherryCrisp: {
  string: "Apricot-Cherry Crisp",      category: 'dessert,price: 1.50},
rhubarbStrawberryCobbler: {
  string: "Rhubarb Strawberry Cobbler", category: 'dessert,price: 1.50},
bakedApple: {
  string: "Baked Apple",               category: 'dessert,price: 1.50},

Water: {
   string: "Water", category:        'beverage, price: 0.95},
coffee: {
   string: "Coffee", category:       'beverage, price: 0.95},
hotTea: {
   string: "Hot Tea", category:      'beverage, price: 0.95},
```

```
        icedTea: {
           string: "Iced Tea", category:      'beverage, price: 0.95},
        milk: {
           string: "Milk", category:          'beverage, price: 0.95},
        soda: {
           string: "Soda", category:          'beverage, price: 0.95},
        dietSoda: {
           string: "Diet Soda", category:     'beverage, price: 0.95},
        lemonLime: {
           string: "Lemon Lime", category:    'beverage, price: 0.95},
        rootBeer: {
           string: "Root Beer", category:     'beverage, price: 0.95},
        lemonade: {
           string: "Lemonade", category:      'beverage, price: 0.95},
        mineralWater: {
           string: "Mineral Water", category: 'beverage, price: 0.95},
     },
}
```

2. After creating this frame, you can then do the following in the Inspector:

```
Print(menu:GetCategories())
["None", "Appetizer", "Soup", "Entree", "Salad",
"Side", "Dessert", "Beverage"]

Print(menu:CategoryToItems("Beverage"))
["Root Beer", "Diet Soda", "Milk", "Coffee", "Lemon-
ade", "Water", "Soda", "Lemon Lime", "Mineral
Water", "Iced Tea", "Hot Tea"]

Print(menu:CategoryToCategorySymbol("Beverage"))
beverage

Print(menu:CategorySymbolToCategory('beverage))
"Beverage"

Print(menu:ItemSymbolToPrice('icedTea))
0.950000

Print(menu:ItemToItemSymbol("Hot Tea"))
hotTea

Print(menu:ItemSymbolToCategory('hotTea))
"Beverage"
```

3. Make sure to save the menu frame code (perhaps copy it to the scrapbook); you'll be need it again in Chapter 7.

Summary

In this chapter we described NewtonScript—the programming language of the Newton. Starting with frames and arrays, you learned about NewtonScript's various data structures and constructs. After covering NewtonScript, we detailed two important features of the language that make it even more excellent: its portability and garbage collection.

You should now have enough information about NewtonScript to begin your Newton programming in earnest. Remember, you can use this chapter as a reference guide as you write your NewtonScript code. Appendix E on page 335 provides syntax definitions for the language as well.

Chapter 6

Inheritance in NewtonScript

We inherit nothing truly, but what our actions make us worthy of.

—*George Chapman*

NewtonScript is an object-oriented language with a unique form of inheritance. This inheritance is integral to the structure of the language and molds how you write your code. This uniqueness and importance require your full understanding before you can create an application or even take advantage of inheritance's key features. Let us look at those key concepts and structure.

Overview of NewtonScript Inheritance

NewtonScript inheritance has two forms: proto inheritance and parent inheritance. While the underlying structure of each is the same, the forms differ in how you use them. Before talking about proto or parent inheritance in particular, let us look at how inheritance works in general.

The Mechanics of Inheritance

Becoming acquainted with the original design requirements of the language goes a long way toward explaining inheritance's structure and why NewtonScript provides a new form of it. The Newton has a limited amount of memory, but a fair amount of ROM (in the form of system ROM and ROM cards). The language, in general, and inheritance, in particular, were molded with this crucial point in mind.

The NewtonScript inheritance model deals with objects. These objects may inherit from other objects. Here is the important factor in the relationship:

- *An object inheriting from another object only contains what is different.*

Difference inheritance is quite unlike the classic model that you would find in standard languages like C++. Before we compare these two models in greater detail, however, let us look at an example of NewtonScript inheritance using two objects: `original` and `copyCat`. If you look in Figure 6.1, you can see that both objects are frames; `copyCat` is inheriting from `original`.

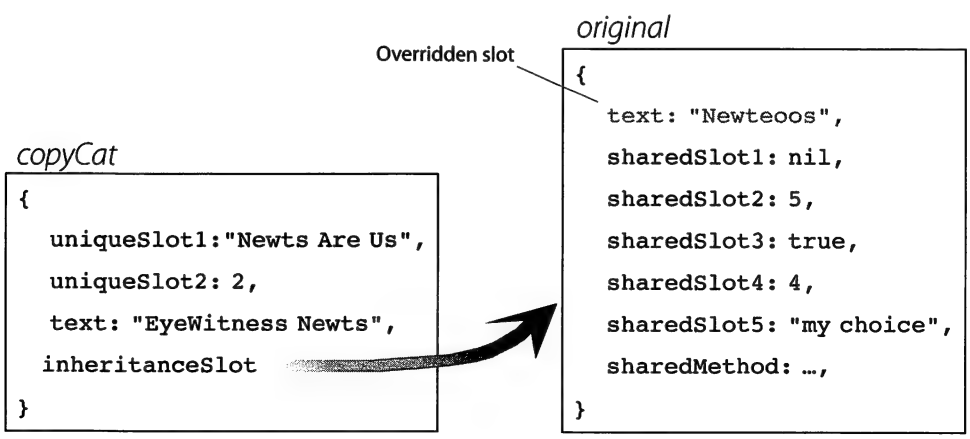


Figure 6.1 One object inheriting from another.

The first object, `copyCat`, contains four slots:

- The first two slots, `uniqueSlot1` and `uniqueSlot2`, are unique to `copyCat`.
- The third slot, `text`, is in both objects.
- The last slot, `inheritanceSlot`, holds the inheritance reference.

The second object, `original`, contains seven slots of its own:

- The first slot, `text`, is overridden by `copyCat`'s own text slot.
- The second through seventh slots contain various values, all of which are available to `copyCat`.

Thus, `copyCat`, whose size is only 4 slots, has access to a total of 10 slots (6 via inheritance). One of `original`'s slots, `text`, is not accessible, as it is overridden.

From this example, you can see that an object can inherit from another object while maintaining unique slots of its own. As in `copyCat`, these unique slots can be any of the following:

New Method	A new slot containing a method. The inheriting object (`copyCat`) can respond to unique messages that the other object (`original`) knows nothing about.
New Data	A new slot containing some data value.
Overridden Method	A method with the same name as an existing method. The existing method is masked by the overridden version (`copyCat`).
Overridden Data	A slot containing data that has the same name as an existing slot. The inheriting object (`copyCat`) has no access to the existing slot.

Actually, there is no distinction in NewtonScript between overriding a method and overriding data. We mentioned these separately to draw your attention to both possibilities. Because many languages don't allow overriding data, this may be unfamiliar to you.

Comparing Prototype Inheritance and Class-based Inheritance

As we said before, NewtonScript inheritance differs from the class-based inheritance found in languages like C++, Smalltalk, and Object Pascal. In class-based inheritance, one class inherits from another class and objects are created as instances of a class.

Difference inheritance has some advantages over class-based inheritance:

- Each object is smaller. In class-based inheritance, each object contains the union of its own data and all the data belonging to its superclasses. This makes for large objects. In difference inheritance, an object only contains a pointer to the object it is inheriting from and its own unique data. This makes for small objects.
- An object can override both methods and data.
- The inheritance is dynamic; you can change what an object inherits from by changing the pointer.
- An object has access to changes made in the object it inherits from and reflects those changes immediately, even at run time.

The major disadvantage is speed. Accessing a slot requires searching an object and then searching its entire inheritance chain. In class-based inheritance, accessing a slot can be done in one operation. Remembering the design requirements of the Newton, the size versus speed trade-off was foreordained.

Proto Inheritance

Now that you understand inheritance by difference we can discuss the first of its two forms: proto inheritance. Proto inheritance is done via a slot named `_proto` that points to the object's proto. Let us revisit in Figure 6.2 our two sample objects to help clarify the relationship between an object (`copyCat`) and its proto (`original`). `copyCat` now contains a `_proto` slot. This slot still points to `original`, which `copyCat` inherits from and which now serves as its proto. We say that `copyCat` *protos from* `original`. The other mechanics of inheritance are still the same; all of `original`'s slots are available, except for `text`, which is still an overridden data slot.

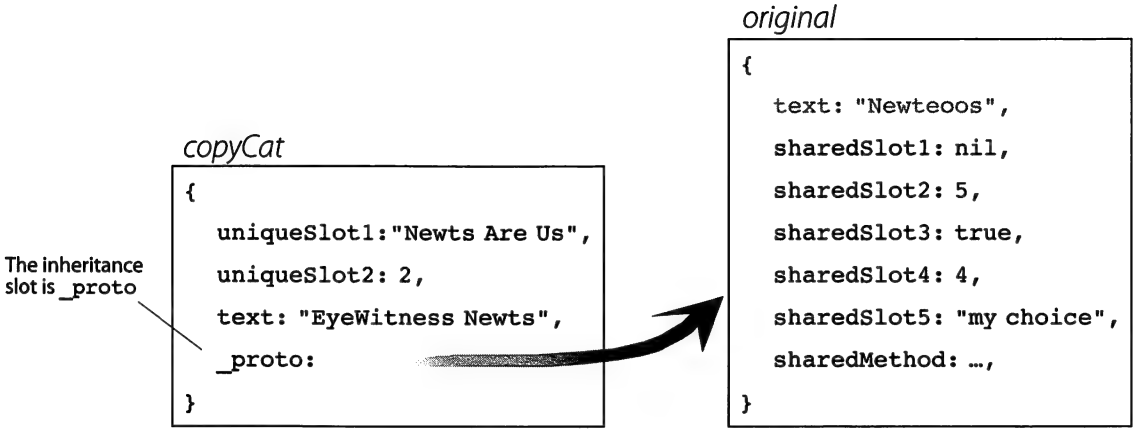


Figure 6.2 One frame protoing from another.

Proto Inheritance and Inheritance Lookup

Imagine you wish to access a slot in a frame, as in this example:

```
copyCat.sharedSlot4 := 69;
```

If the slot can't be found in the frame, NewtonScript then looks in the frame's proto for the slot. If the slot is still not found, then each proto in the proto chain is searched until either the slot is found, or the last proto (without a `_proto` slot) is searched. For example, this code:

```
copyCat.faroff := 6;
```

would finally find the slot, but not until the entire proto chain had been searched (see Figure 6.3).

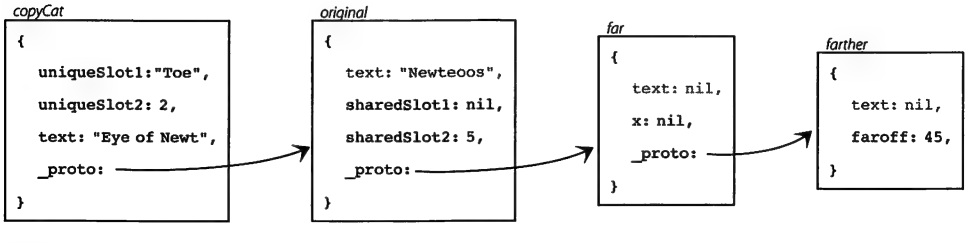


Figure 6.3 Searching the entire proto chain to find a slot.

Multiple Protoing

Multiple objects can also proto from the same proto source (see Figure 6.4). Each object has its own unique values and inherits all of the proto's slots as well.

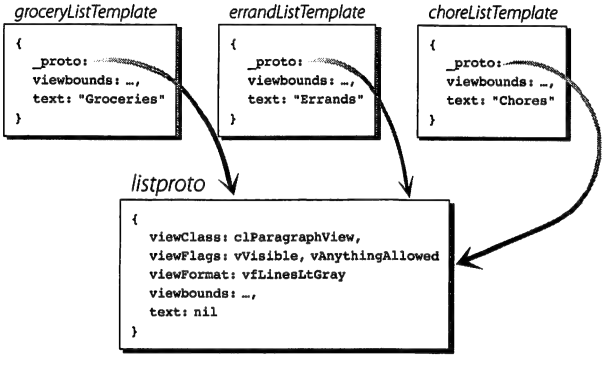


Figure 6.4 Three objects inheriting from the same proto.

Slot Assignment

It is crucially important to understand how inheritance deals with an assignment statement to an inherited slot. *Assigning to an inherited slot never changes the proto.* There are two compelling reasons for this:

- Objects should be independent.
- Protos may be in ROM.

Objects Should be Independent

In cases where multiple objects inherit from the same proto, a change to one of those objects should not affect any of the others. For example, in Figure 6.4, you could assign a new value to the `viewFormat` slot of `groceryListTemplate`:

```
groceryListTemplate.viewFormat := // black lines
```

You would not want this assignment to affect the way the lines look in `errandsListTemplate` or `choreListTemplate`. To change the `viewFormat` slot in all templates, you would change the `viewFormat` slot in `listproto` itself.

Protos May Be in ROM

An obvious example, system protos, are most certainly in system ROM and thus are read-only. The rest of your application's protos are in read-only form as well. Protos can only be in one of these places:

- on a ROM card
- on a write-protected RAM card
- on a non write-protected RAM card
- in internal memory

Even in the latter two cases, however, the Newton uses its memory management unit (MMU) to disallow writing to application memory. The only writable portion of your application is its views. For all intents and purposes, your application's protos and templates are not writable, so we say that they are located in ROM or pseudo-ROM.

Assignment to Inherited Slots

If you cannot write to an inherited slot with an assignment, what does happen? Pretty much what you would expect—a new slot is created. The new slot is assigned the value and is placed in the inheriting frame. Thus, the easy example of `copyCat` and `original` and this assignment:

```
copyCat.sharedSlot4 := 69;
```

creates a new slot in `copyCat` with the value of `69` (see Figure 6.5).

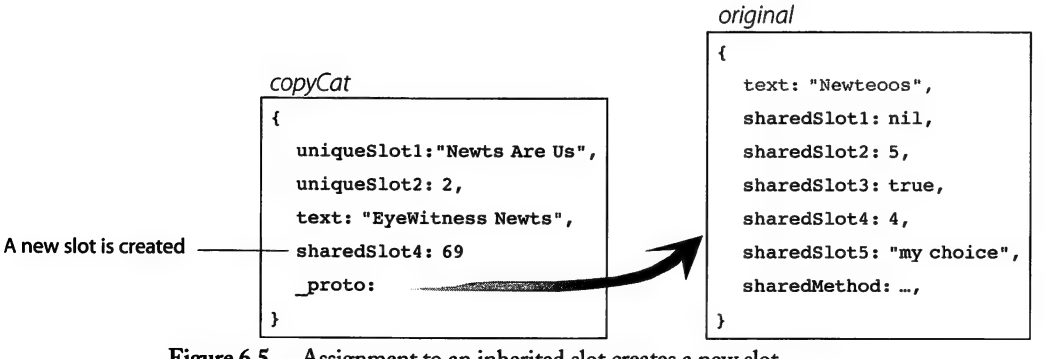


Figure 6.5 Assignment to an inherited slot creates a new slot.

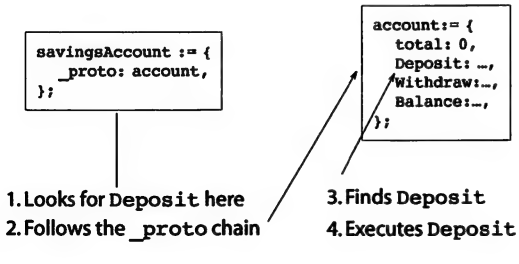
Now, let us examine a more interesting example. Here we have a frame `account`, and two others frames, `savingsAccount` and `checkingAccount`, that proto from it:

```
account:= {
   total: 0,
   Deposit: func(amount)
            begin
               total := total + amoun
            end,
   Withdraw:func(amount)
            begin
               total := total - amount;
            end,
   Balance: func()
               return total,
};

savingsAccount := {
   _proto: account;
};

checkingAccount := {
   _proto: account;
};
```

We will send some messages to `savingsAccount` and `checkingAccount` and see what happens to the slots in our three frames:

```
savingsAccount:Deposit(2000);
checkingAccount:Deposit(500);
savingsAccount:Withdraw(200);
```

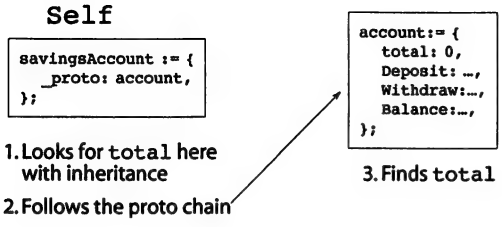
Before sending the messages, `savingsAccount` and `checkingAccount` only have one `_proto` slot each.



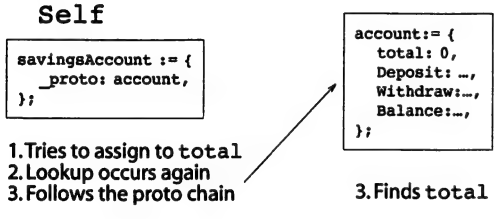
When the `Deposit` message is sent to `savingsAccount`:

```
savingsAccount:Deposit(2000);
```

the run-time system looks for the `Deposit` method within the `savingsAccount` frame. Since it isn't there, it follows the proto pointer to `account`. It finds the `Deposit` method there and executes it.



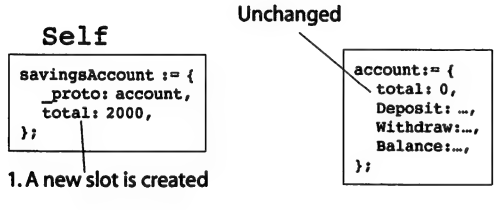
While `Deposit` is executing, the value of `self` is `savingsAccount` (`self` is the frame that receives the message). When `Deposit` attempts to access the current value of `total`, lookup begins. `total` is looked for first as a local variable, then using inheritance rules. Inheritance lookup starts with `self` and then follows the proto chain. `total` is found in account, with a value of `0`.



When the `Deposit` method:

```
Deposit:func(amount)
begin
 total := total + amount;
end;
```

tries to assign to `total`, the lookup takes place again—starting first in `self` and then following the proto chain. `total` is found in account.



Notice that `total` in `account` is not changed; instead, a new `total` slot with a new value is created in `self` (`savingsAccount`). This overrides the `total` slot in `account`.

When the `Deposit` method finishes, `savingsAccount` contains two slots, a `_proto` and a `total` slot. After you execute the next message to `checkingAccount`, the same process occurs all over again. When you have finished, `checkingAccount` will, likewise, have two slots, a `_proto` slot and a `total` slot. When you execute the third message:

```
savingsAccount:Withdraw(200);
```

`savingsAccount` will have no new slots, but the value of its `total` slot will now be `1800`.

The inherited Keyword

How do you write an overridden method that augments the original, rather than completely replacing it? You use the `inherited` keyword:

```
frame.Method := func(a)
begin
   // do some stuff

   // and then call old version
   inherited:Method(a);
end;
```

When you use `inherited` to call an old version of a method, `self` retains its current value. As far as the old method is concerned, it is as if the overridden method were not there (`self` remains the same).

The :? Operator

In some cases you may provide a method and not know whether a previous version exists. For example, you may want to use a view system method in a template, and not know whether its proto contains that method. NewtonScript provides a solution to this problem in the form of the `:?` operator:

```
inherited:?Method()
```

This calls the inherited version of `Method` only if it exists. If not, it does nothing. It is equivalent to having written:

```
if inherited:Method exists then
      inherited:Method();
```

It is quicker when an inherited version of `Method` exists, however, as lookup occurs once rather than twice.

If you are trying to determine whether a method exists, make sure to use *`frame:Method`* `exists` and not *`frame.Method`* `exists` (colon not period). The former uses both proto and parent inheritance; the latter only uses proto inheritance.

The deeply Version of foreach

The `foreach` loop normally iterates over each slot in a frame. In some cases, you may want to iterate not only the slots that are in a frame, but the slots that the

frame inherits. The `deeply` version of foreach does exactly that. It iterates over each slot, including inherited slots, but skips the `_proto` slot.

Here is an example that shows the difference:

```
foo := {a: 3, b: 6};
bar := {_proto: foo, a: 5};

foreach name, value deeply in bar do begin
   Write(name & ": ");
   Print(value);
end;
a: 5
a: 3
b: 6

foreach name, value in bar do begin
   Write(name & ": ");
   Print(value);
end;
_proto: {
   a: 3,
   b: 6}
a: 5
```

In foo

In bar

In foo only

Parent Inheritance

Now it is time to cover the second form of inheritance in NewtonScript: parent inheritance. An object can have two inheritance slots, one for its proto and one for its parent. The `_parent` slot points to another frame from which the object inherits and is used only in these two instances:

- To find a variable when no explicit frame is given.
- To determine what method to execute when a message is sent.

Let us look at this first instance by examining the following code:

Parent inheritance not used

```
copyCat.unknownSlot := "Grin";
someSlot := 72;
```

Proto inheritance is used both for looking up a variable and for explicitly accessing a slot from a frame. Parent inheritance is not used when explicitly accessing a slot from a frame.

When you use slot access (like `copyCat.unknownSlot`) then only `copyCat` and its proto chain are searched for `unknownSlot`.

On the other hand, if you do not use dot syntax (like `someSlot := 72`), then lookup occurs in the following order:

- Search for `someSlot` as a local variable.
- Search for `someSlot` as a global variable.
- Search in `self`, and `self`'s proto chain.
- Search in `self`'s parent, and its proto chain. Continue through each ancestor of `self` and each ancestor's proto chain.

The other time that parent inheritance comes into play is when you send a message to a method. So, for instance:

```
copyCat:SomeMessage(felix)
```

will search the parent chain of `copyCat` to find the `SomeMessage` method.

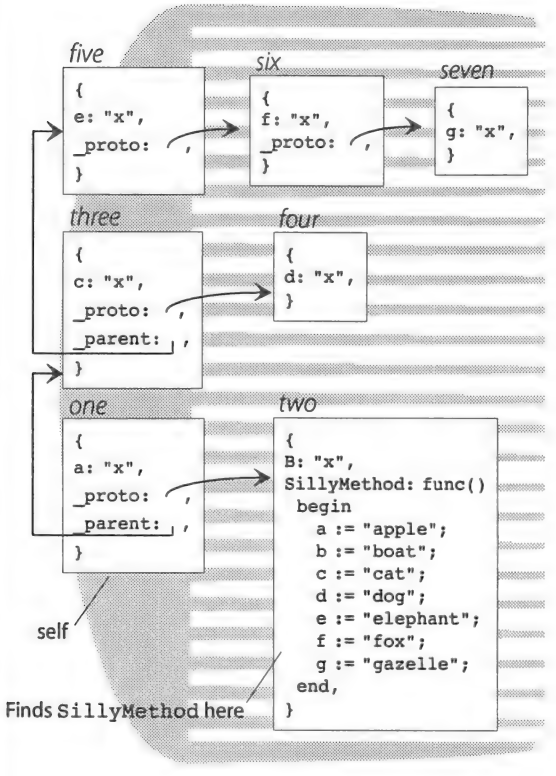
Now that you have that brief introduction to parent inheritance let us look at how proto and parent inheritance combine with each other in NewtonScript.

Combining Proto and Parent Inheritance

It is easiest to think of a comb when you wish to visualize how proto and parent inheritance work together. Here are the basic rules of assignment when inherited slots are involved:

- A child proto chain is fully searched before searching its parent's proto chain *(one tooth of the comb at a time)*.
- Assignment only changes slots along the spine of the comb.
- The level at which a slot is found is the level at which the assignment will be made.

Now let us look at a diagram of several frames that have `_parent` and `_proto` slots in them. One of the frames has a method, `SillyMethod`, that makes assignments to slots in all of the frames.



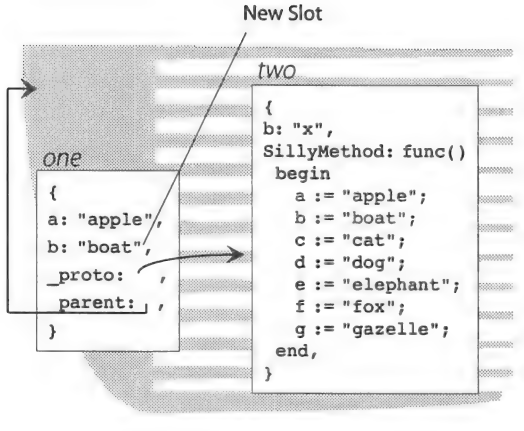
Let us walk through what happens when `SillyMethod` gets executed. The message we will send is:

```
one:SillyMethod();
```

The current function is searched for a local called `SillyMethod`. Next, inheritance rules come into play: `one` is searched for `SillyMethod`. When this fails, lookup follows the proto chain to `two` and searches for `SillyMethod` there. When lookup finds it, it sets `self` to `one`, and `SillyMethod` is executed and its first assignment is made:

```
a := "apple";
```

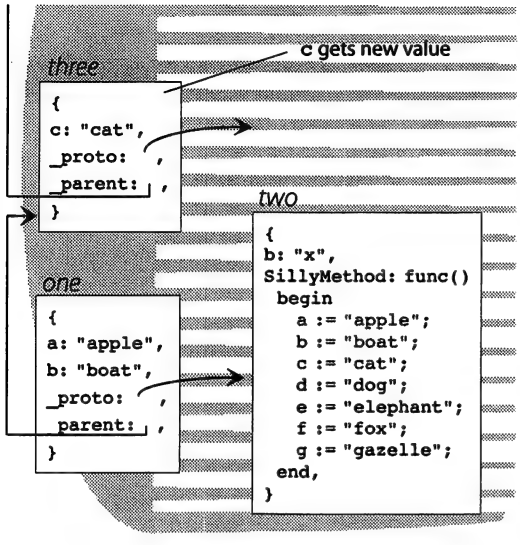
`a` is first looked up as a local in `SillyMethod`, and then as a global. Then it is searched for in `self`. When `a` is found it gets assigned its new value right there in its frame, `one`.



The next statement in `SillyMethod`:

```
b := "boat";
```

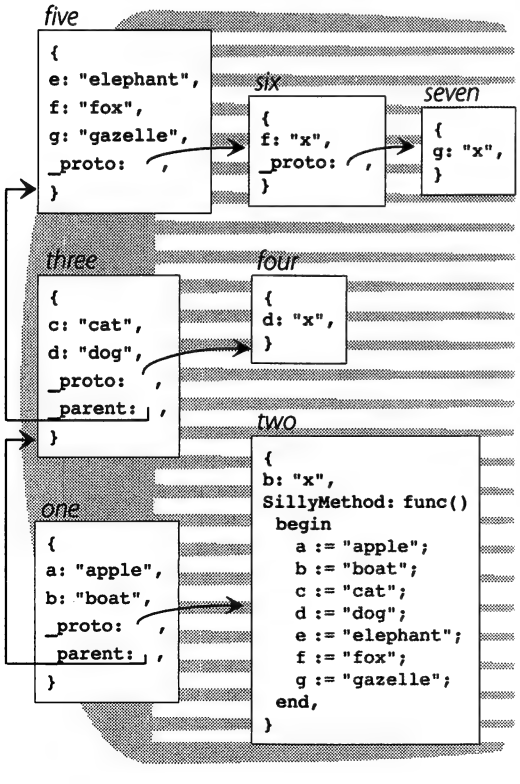
gets executed and lookup begins for `b`. First, it is looked up as a local in `SillyMethod`; then it is looked up as a global. Using inheritance, `one` is searched for `b`, and then lookup follows the proto chain to `two`. This frame contains a `b`, so lookup is finished. To make a successful assignment requires creating a new slot with the value of `"boat"` in `one`. *Remember, an assignment can never modify the proto.*



`SillyMethod` can now execute the third assignment:

```
c := "cat";
```

Lookup begins for `c`. `SillyMethod` is searched first for a local `c`, and then `c` is looked up as a global. Using inheritance, `one` is searched for `c`, and then lookup follows the proto chain to `two`. Now lookup, having exhausted the proto chain, starts up the parent chain in the quest for `c`. This frame contains a `c`, so lookup is successful. The `c` slot in `three` now gets assigned the new string value.



`SillyMethod` can now execute the fourth assignment:

```
d := "dog";
```

Lookup begins for a local `d` in `SillyMethod`, and then a global `d`. Using inheritance, `one` is again searched for `d`, and then lookup follows the proto chain to `two`. Not finding the slot in the proto chain, lookup starts up the parent chain in its quest for `d`. `three` does not contain `d` so lookup continues out the proto chain to `four`. This frame contains a `d` slot, so lookup is successful. Assignment to `d` requires creating a new slot with the value of `"dog"` in `three`. *You do not modify the proto, and assignment occurs at the level at which a slot is found.*

The lookup occurs in the same way for the last three assignments to `e`, `f`, and `g`, though one more level up the parent chain. At this point, you will have new and modified slots on the comb's spine.

Accessing Slots

There are also slightly differing ways to access slots from a frame. Table 6.2 shows these various ways. Note that they differ in whether or not you use a symbol to access the slot, and in their combinations of proto and parent inheritance.

Notice that there is a definite difference between `slot` and `self.slot`. Using `self` disables parent inheritance. Only use it when you are certain you don't want parent inheritance. "Using self in a Method" on page 164 covers this issue in more detail.

Syntax	*Example*	*Uses Proto Inheritance*	*Uses Parent Inheritance*
`slot`	`slotA`	X	X
`frame.slot`	`self.slotA`	X	
`frame.(pathExpr)`	`self.('slotA)`	X	
`GetVariable(frame, symbol)`	`GetVariable(self, 'slotA)`	X	X
`GetSlot(frame, symbol)`	`GetSlot(self, 'slotA)`		

Table 6.1 Different ways to access slots.

Testing for the Existence of Slots

There are also a number of different ways to test whether a slot exists. Table 6.2 summarizes these methods.

Syntax	*Example*	*Uses Proto Inheritance*	*Uses Parent Inheritance*
`slot exists`	`slotA exists`	X	X
`frame.slot exists`	`self.slotA exists`	X	
`frame.(pathExpr) exists`	`self.('slotA) exists`	X	
`frame:slot exists`	`self:MethodA exists`	X	X
`HasVariable(frame, symbol)`	`HasVariable(self, 'slotA)`	X	X
`HasSlot(frame, symbol)`	`HasSlot(self, 'slotA)`		

Table 6.2 Different ways to test for the existence of slots.

Inherited

When you use the keyword `inherited` for a method, lookup will only search the proto chain. If you have overridden a method from your parent, and attempt to use `inherited` to call the parent version, you'll get a run-time error since `inherited` never searches in the parent.

Sending the same message to your parent will call the overridden method, but the value of `self` will change from its current value to be the parent frame. This may cause the parent method to operate differently. A different `self` can have some effects you will not appreciate.

Using self in a Method

Here are rules of thumb for using `self` within a method:

- Don't use `self` when sending yourself a message. Use `:Message()`, not `self:Message()`.

Rationale: Although the meanings of the two are identical, it is confusing for the reader. `self:Message()` looks very similar to `self.slot`. They operate differently since `self:Message()` follows parent inheritance, but `self.slot` does not. Things that look similar but work differently invariably cause confusion.

- Don't use `self.` when reading from a slot unless you know the slot is in your proto chain.

Rationale: `self.slot` doesn't use parent inheritance, so NewtonScript won't ever find it if it is up there.

- Now a twist on the last: as long as a slot is not in a parent, use `self.` when assigning.

Rationale: If you don't use `self.`, and the slot does not exist in the proto chain, you will end up with a local variable being created instead of a slot. The explicit `self.slot` means that `slot` will be created in `self` whether or not it exists in the proto chain.

- When you write methods that create slots in `self`, also create your slots at compile time and set their values to `NIL`.

Rationale: If you use code like `slot := value` (instead of `self.slot := value`), `slot` will be found in the proto chain, and `slot` will be created in `self`. If you

don't create `slot` at compile time, `slot` won't be found in the proto chain and will be created as a local variable instead.

Although this rule of thumb isn't necessary if you follow the previous rule of thumb (using `self.slot := value`), the two rules are an example of the belt-and-suspenders approach to life. Even if you forget one, the other will keep you from harm.

Don't Use _parent

It will seem natural to write code using the `_parent` pointer much as you use the `_proto` pointer. But it does not work as you intend. Instead, you'll usually use `:Parent()`, a view message that returns the parent view. If you happen to be writing methods for nonview objects, use can use `self._parent`.

Note: You cannot use `_parent` because of a quirk in the current NewtonScript run-time implementation. The runtime itself uses inheritance. As your method is executing, `_parent` and `self` both point to the frame that received the message. Using `self._parent` retrieves the `_parent` slot from the correct frame.

Use Parent Inheritance for Inheriting Data

In general, you should reserve parent inheritance for inheriting shared data. It is unusual to use it to inherit behavior. Here is the rationale:

- If you use parent inheritance to inherit behavior, `self` is a child view while the parent is where the behavior is found. Many methods expect `self` to be somewhere in their proto chain.

For example, the method might assign to `self.slot`, which creates a slot in the child. Or, the method might iterate over its children.

This means that if you have a hierarchy, such as in Figure 6.6, to call the `MethodA` from inside `MethodB`, you should call `:Parent():MethodA()`, and not `:MethodA()`. The former sends a message to the parent object; the latter sends a message to the child object.

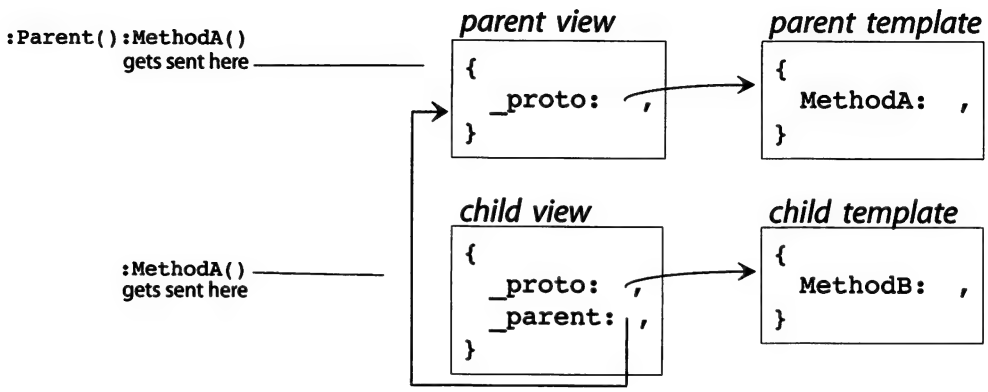


Figure 6.6 A hierarchy of frames.

NewtonScript, Newton Toolkit, and the Newton

Now it is time to tie everything together: NewtonScript, NTK, inheritance, and the Newton. To do this, we need to return to the comb diagram and talk about the differences between views, templates, and protos in relationship to inheritance and assignment (see Figure 6.7).

Newton Run-time Views

All of the frames along the spine of the comb are views. These views are created at run-time, and they are the only frames that can be written to at run-time; whenever assignments are made, the relevant slots are created in the views. Views are the receivers of messages and contain both `_parent` and `_proto` slots.

The `_proto` slot of a view always points directly to its template, which was created in NTK (there may be more protos after that in the chain). The `_parent` slot of a `protoApp` view points to the root view on the Newton.

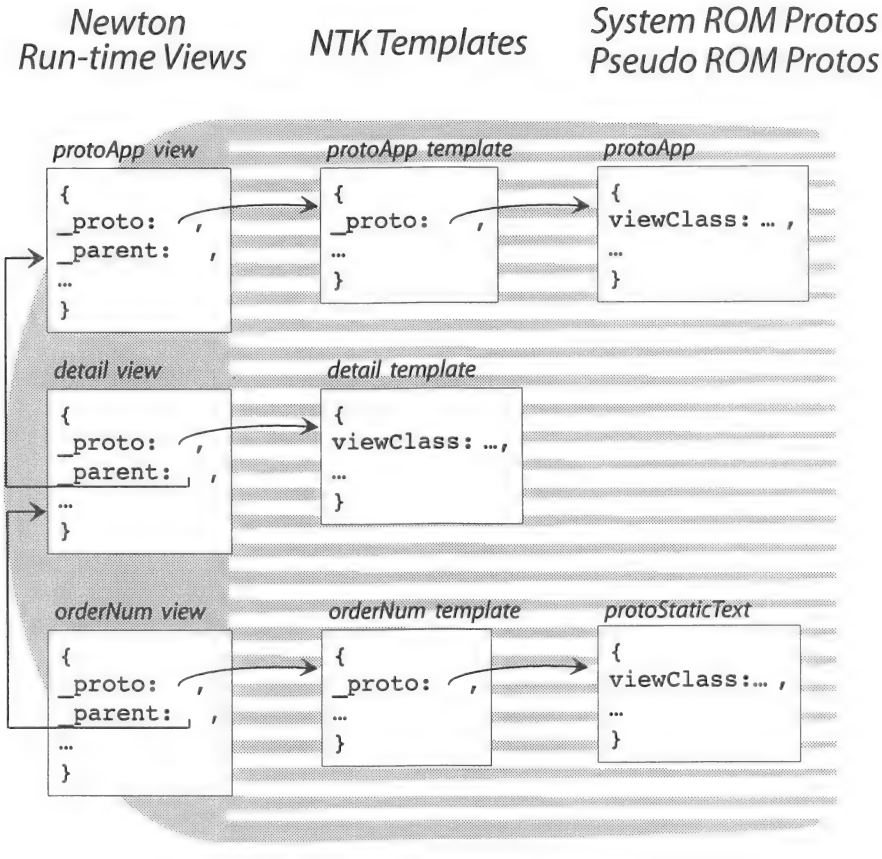


Figure 6.7 Views, templates, and protos in ROM.

NTK Templates

You create your templates in NTK. These templates do not display `_parent` slots, but they do have `_proto` slots. The parent–child relationship that the views will use is determined by the order you lay out the templates in the layout window. These templates either will have `_proto` slots that point to protos in ROM or will have a `viewClass` slot.

Seeing the _parent Slot

We we said previously, although you see `_proto` slots in NTK, you do not see `_parent` slots. This is because templates and protos don't contain `_parent` slots; only views do. For those templates that have children, NTK adds a `stepChildren` slot, which is an array of templates. At run time, the view system reads that slot and creates child views based on the slot. The view system then sets up the `_parent` slot in the view. This topic is discussed in complete detail in "How Views Are Created" on page 170. You will not see the `stepChildren` array in the browser either. NTK creates the array as it builds the application.

Summary

This chapter covered NewtonScript's unique form of inheritance. You learned that this inheritance is by difference: objects only store what is different from the object they inherit from. Next, we covered the two forms of inheritance: proto inheritance and parent inheritance. We also did a detailed walkthrough of assignment and inheritance lookup rules. You should now have a clear idea of how assignment and slot access work in NewtonScript. We then covered some of the uses of self and how to send messages to your parent. Last of all we looked at the relationship between inheritance, the Newton, NTK, and protos.

Heady stuff, this.

Chapter 7

View System and Messages

All objects lose by too familiar a view.

—*John Dryden*

Views exist only on the Newton. A view itself is just a frame composed of:

- `_parent` and `_proto` slots
- a `viewCObject`—an object in the underlying C++ view system that does the actual drawing of the view
- transitory data

A number of different messages are sent by the Newton view system to views. In each case, the message is sent to the view, and the method is found via proto inheritance. First, let us explore how these views are created on the Newton, and then we'll describe the various view methods you will use in your application.

How Views Are Created

Views proto from a template (or sometimes directly from a proto) and contain an integer `viewCObject` slot. The `viewCObject` is a C++ object in the underlying view system. It handles the actual drawing, handwriting recognition, displaying pictures, etc.

When your application is launched it is sent an `Open` message. Its view's are then sent these three messages in this order:

- `viewSetupFormScript`
- `viewSetupChildrenScript`
- `viewSetupDoneScript`

Before going into the details of each, let us look at what happens on the Newton when your application's views get opened. As you can see in Figure 7.1, the first thing that happens is the view system creates an empty view frame (unless it already exists). Next, the `viewSetupFormScript` is executed and it adds or modifies any necessary slots in the view. After this, the `viewCObject` is created by reading from slots in the view.

The next message that is executed is the `viewSetupChildrenScript`, which modifies the `stepChildren` array, if necessary. If the view has any children, the view creation is started again from scratch for the first child view in the `stepChildren` array (see the second column in Figure 7.1). If the child has any children of its own (the grandchildren), they are created after the child's own `viewSetupChildren` script executes.

Once the last child (one with no children of its own) is created, that child's `viewSetupDoneScript` gets executed (see the third column in Figure 7.1). The view system winds itself back up the ancestor chain, executing the `viewSetupDoneScript` of each view until it reaches the application's base view (the first view in the chain). Presto! Your views are now open on the Newton.

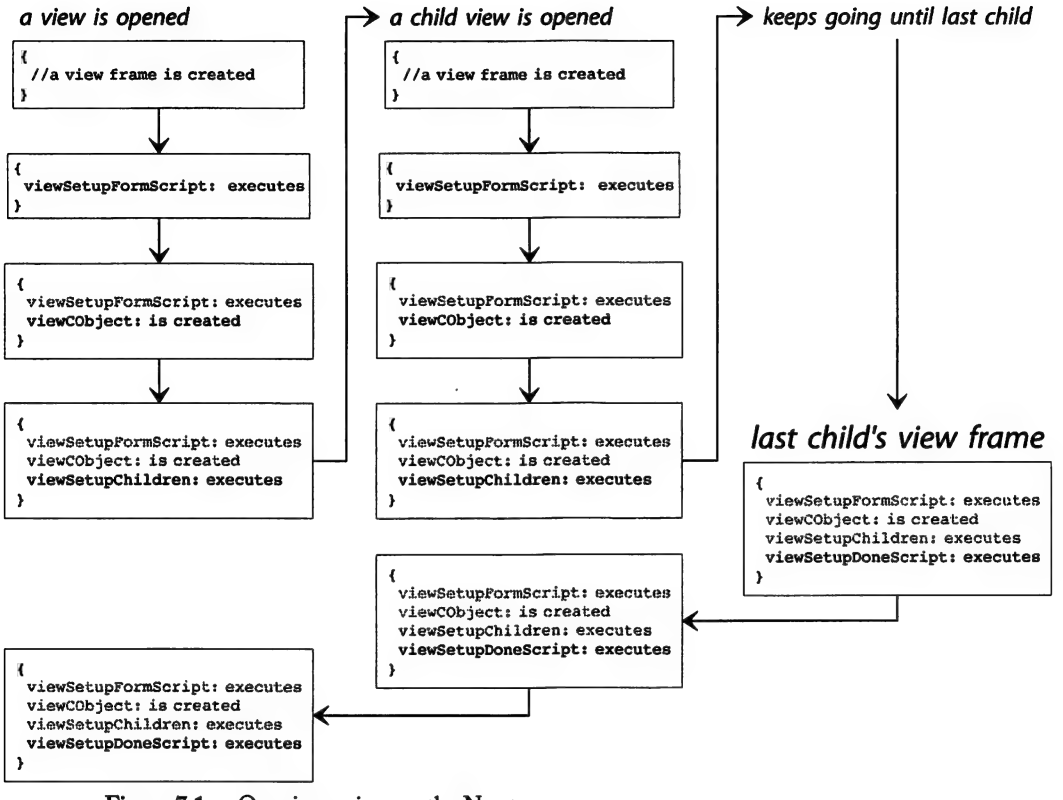


Figure 7.1 Opening a view on the Newton.

As you can see, the same three messages are sent to each view in exactly the same order. You will handle particular things in each message. Let's look at each message in turn.

viewSetupFormScript

The `viewSetupFormScript` message is sent *before* the `viewCObject` of a view is created. A number of the `viewCObject`'s aspects are determined when it is created by slots in the view. Because of this, the `viewSetupFormScript` is the only place you can modify those slots and have the `viewCObject` notice. For instance, `viewBounds`, `viewJustify`, and `viewFormat` are some of the slots that the `viewCObject` reads from the view.

It follows that if you want to modify the values of those slots, this is the place to do it. The `viewSetupFormScript` can override slots in the template. This ensures that the `viewCObject` reads the overridden slots rather than the template's slots. Here are some of the more typical aspects of a view that you use `viewSetupFormScript` to modify at run-time:

- The `viewBounds` slot—to set the size of the view
- The `viewJustify` slot—to set the justification of the view
- The application's size—to account for various screen sizes
- Application initialization code (for example, to create soups)

Setting Justification and Bounds in viewSetupFormScript

Here is an example of setting the bounds of a view at run-time, rather than from within NTK. This is the `viewSetupFormScript` for the `justifiableProtoLabelInputLine`—the one that correctly handles horizontal `parentRelativeFull` justification:

```
func()
begin
   // doesn't yet handle sibling justification
   // if there is a viewSetupFormScript, call it
   inherited:?viewSetupFormScript();

   local justify := viewJustify;

   // if no viewJustify, the default is top-left
   if justify and bounds then begin
      local horizontalParentJustify :=
         Band(justify, vjParentHMask);
      local horizontalSiblingJustify :=
         Band(justify, vjSiblingHMask);
      //ChildViewFrames returns an array of children
      local siblings :=
         :Parent():ChildViewFrames();
      local numSiblings := Length(siblings);

      if horizontalSiblingJustify = vjSiblingNoH or
         numSiblings = 0 then begin
         if horizontalParentJustify = vjParentFullH
            then begin
            // parentBounds.left is 0
```

Subtle point

```
                    local parentRightBounds :=
                       :Parent():LocalBox().right;
                    local bounds := Clone(viewBounds);
                    bounds.right := bounds.right +
                       parentRight;
                    self.viewBounds := bounds;

                    // turn on only left justify
                    justify := Bor(BAnd(justify,
                       BNot(vjParentHMask)),
                       vjParentLeftH);
                    self.viewJustify := justify;
                    end;
            end;
         end;
      end
```

Basically, the code determines whether `parentRelativeFull` horizontal justification is requested and determines whether sibling justification should be used as well. Once it determines that `parentRelativeFull` needs to be used, the code modifies the `viewBounds` to the correct value for `parentRelativeLeft` justification and sets the horizontal justification to that. When the `viewCObject` reads `viewJustify` and `viewBounds`, it sees a simple `parentRelativeLeft` justification with appropriate bounds.

Assigning to viewBounds Correctly

There is one subtle point in the above code: the portion that modifies `viewBounds`. You might have been tempted to write it like this:

```
viewBounds.right := viewBounds.right + parentRight;
```

The problem with this assignment is that `viewBounds` is a read-only frame that is still part of a template. This assignment tries to follow the proto chain back to `viewBounds` in the template and modify it there (see the left side of Figure 7.2). It does you no better to try this code either:

```
viewBounds := viewBounds;
viewBounds.right := viewBounds.right + parentRight;
```

This creates a new slot in the view, but that slot simply points to the same `viewBounds` frame as the template (see the right side of Figure 7.2). This is still attempting to write to read-only memory.

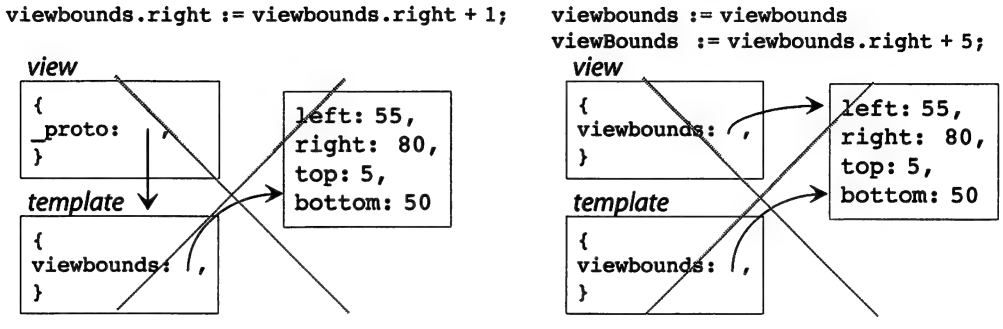


Figure 7.2 The wrong way to set viewBounds.

To set the viewbounds correctly, you must create a whole new frame. First is some code that clones the viewBounds frame and assigns it to a local bounds. Next, a new viewBounds slot is created in the view with the code:

```
self.viewBounds := bounds;
```

Some slot must point at the new frame, and it can't be the viewBounds slot in the template (the template is in pseudo-ROM and can't be changed). So, a viewBounds slot is added to the frame (see Figure 7.3).

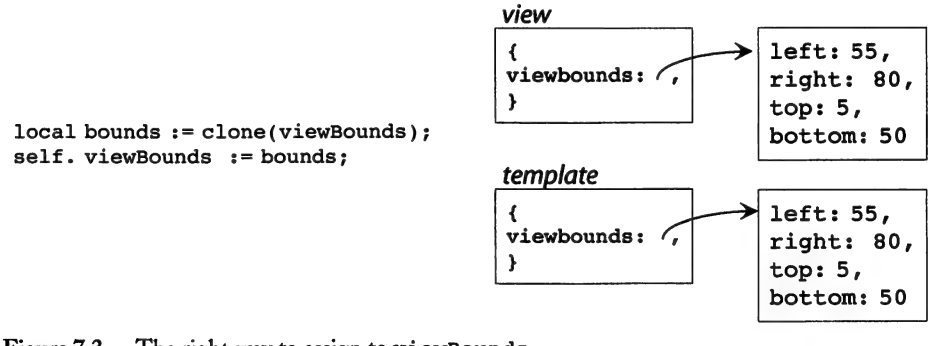


Figure 7.3 The right way to assign to viewBounds.

Setting the Application Size at Run-time

Another use of the viewSetupFormScript is to set the size of the application base view. For information on that use, see "Setting Application Bounds Based on the Screen Size" on page 343.

When your application is opened, the first method that is called is your base view's `viewSetupFormScript`. Therefore, application initialization code is commonly located there.

viewSetupChildrenScript

The `viewSetupChildren` method is called after a view's `viewCObject` is created, but before any child views are created. This makes it a logical candidate for run-time adjustments to the number of children a view will have. You cannot individually modify the children, however, as the child views have not yet been created (all that exists is read-only templates).

stepChildren and viewChildren

The Newton view system creates child views based on the contents of the `stepChildren` and `viewChildren` arrays. These two slots are arrays of templates. NTK creates a `stepChildren` array for each template that has children; system protos have `viewChildren` slots to contain their child templates.

Right after `viewSetupChildrenScript` executes, the view system reads the `stepChildren` and `viewChildren` slots and creates an associated view for each template found there.

Creating Children Dynamically

To create children dynamically, you would modify the `stepChildren` array in your `viewSetupChildrenScript`. For example, in the overview of the WaiterHelper application, the number of displayable rows depends on the height of the overview, which in turn depends on the size of the screen. Rather than statically determining the number of children at compile time, the decision can be delayed until runtime when the screen size is known. Here is a `viewSetupFormScript` for the overview template which creates just the right number of children:

```
func()
begin
   local viewHeight := :LocalBox().bottom;
   local protoHeight := pt_row.viewBounds.bottom;
   local numRows := viewHeight div protoHeight;

   self.stepChildren := Array(numRows, pt_row);
end;
```

This code creates a `stepChildren` array slot in the overview view. Each entry in the array points to the row proto (`pt_row` is the row proto; see "Referring to User Protos from Your Code" on page 100). When the views are created, they will point directly at the proto, rather than to an intermediate template. This is fine, since the templates for the row had no slots other than a `_proto` slot anyway.

viewSetupDoneScript

The `viewSetupDoneScript` message is sent to a view after all its children have been completely created (see Figure 7.1). This message gives the view an opportunity to access children slots or send them messages.

Here is an example of a `viewSetupDoneScript` message located in the application base view. It will open one of two views, depending on which was last open:

```
func()
begin
   // overviewWasOpen is a boolean slot set
   // by the viewQuitScript based on which view
   // was open when the application was closed.
   if overviewWasOpen then     // this slot
      overview:Open();
   else
      detail:Open();
end;
```

viewShowScript

The `viewShowScript` is called when a view is opened or shown.

How Views Are Destroyed

When a view is sent the `Close` message, the `viewCObject` for that view is destroyed (along with the viewCObjects' of all descendants). In addition, the view and each of its children are sent the `viewQuitScript` message. The views for all descendants are destroyed, as is the view itself (except for views declared in some other open views). If you study Figure 7.4, you will get a clear idea of how the view system deals with the destruction of the hierarchy of views.

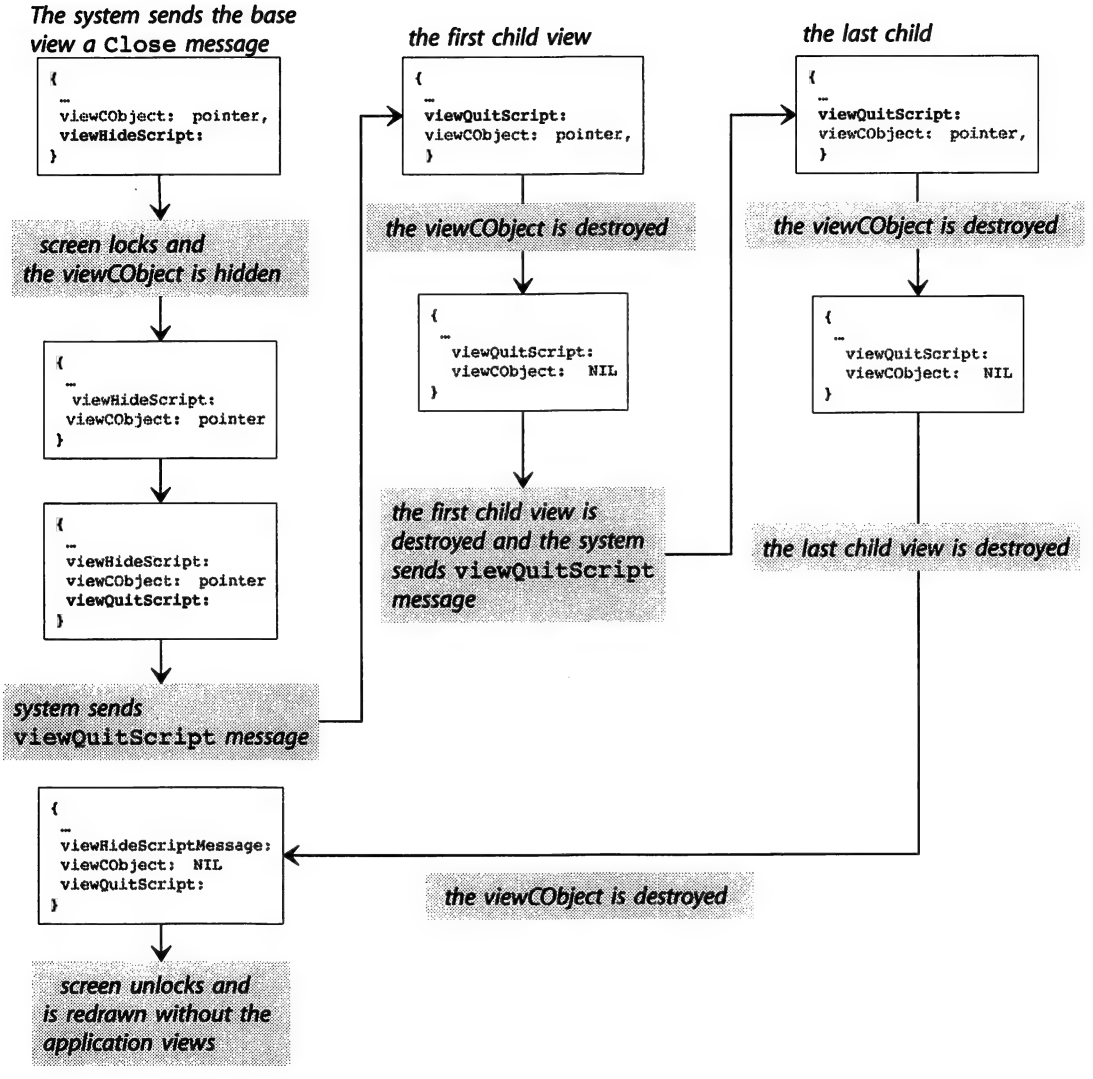


Figure 7.4 How views are destroyed.

viewQuitScript

This message is called when a view (or one of its ancestors) receives the `Close` message. When the method is called, the view still exists, but its parent, or children may not exist. The order in which `viewQuitScript` messages are sent to a

view hierarchy is undefined. The specific view that was `Closed`, however, receives the `viewQuitScript` message before any of its descendants.

viewHideScript

This message is sent when a view is hidden.

Other Messages the View System Sends

viewOverviewScript

The `viewOverviewScript` message is sent to the frontmost view that has the `vApplication` bit set. This message is sent when the user clicks on the overview button.

viewScrollUpScript

The `viewScrollUpScript` message is sent to the frontmost view that has the `vApplication` bit set. This message is sent when the scroll up arrow is tapped.

viewScrollDownScript

The `viewScrollDownScript` message is sent to the frontmost view that has the `vApplication` bit set. This message is sent when the user clicks on the scroll down arrow.

View Messages You Send

Caution: Don't override any of these methods. They are intended to be used as is.

```
Close()
```

Sending a view the `Close` message destroys its `viewCObject`, after which it is no longer displayed. The view itself is also destroyed unless it is declared to a currently open ancestor.

`Open()`

Sending a view the `Open` message causes it to open. This involves creating a `viewCObject` for the view and appropriately dealing with all of its children (creating views and viewCObjects).

No error occurs when you send this message to an open view, though nothing happens.

`Close()`

Sending a view the `Close` message destroys its `viewCObject`, after which it is no longer displayed. The view itself is also destroyed unless it is declared to a currently open ancestor.

`Hide()`

Sending a `Hide` message removes the on-screen display of a view. The `viewCObject` continues to exist, but it no longer draws the view. The `viewHideScript` of the view is called to hide the view.

`Show()`

The opposite of `Hide` is `Show`. Sending a view this message causes it to display on-screen. If the view had been hidden, it redisplays. If the view had not been open, it is opened. The `viewShowScript` of the view is called to make it display.

`Hilite(`*`turnOn`*`)`

Sending a view the `Hilite` message causes it to either highlight (if *`turnOn`* is non-`NIL`) or to unhighlight (if *`turnOn`* is `NIL`).

`Dirty()`

Sending a view the `Dirty` message gets it redrawn by the view system at the next idle time.

`LocalBox()`

The `LocalBox` method returns a frame containing the current bounds of the view. The bounds are in coordinates local to that view (the top-left is (0, 0)). This differs greatly from accessing the `viewBounds` slot. Remember, the `viewBounds` slot is relative to the justification specified in `viewJustify`.

`GlobalBox()`

The `GlobalBox` method returns a frame containing the current bounds of the view. The bounds are in screen coordinates (the top left of the screen is (0, 0)).

`ChildViewFrames()`

This method returns an array of child views. Use this rather than using the `stepChildren` array, since the `stepChildren` array is an array of templates, whereas `ChildViewFrames` returns an array of views.

Declaring Views

In NewtonScript, children inherit from parents, but parents don't inherit from children (like real life).

Parent inheritance gives children access to slots in their ancestors—they can send messages to ancestors or access their data. Parents, on the other hand, can't easily access slots from or send messages to their children.

Of course, you know how to access a child from a parent by using the `ChildViewFrames` message (this returns an array of children). But these children do not have names so they cannot get messages. (It's as if the parents never named their children, referring only to them as the 1st child, the 2nd child, and so on.)

Declaring a view to its ancestor provides a name for the child that can be used by that ancestor. Declaring a child view to an ancestor creates a slot in that ancestor whose value is the child view. Having done this, the ancestor can access slots from and send messages to the child using the child's name.

Note: This is a straightforward operation in NTK and is described in "Using Declare To on a Template" on page 366.

Declaring a child to an ancestor in NTK does *not* create the slot with the child's name in the template. It is not until you get to the Newton, where views exist, that such named slots exist. What NTK does do is create the slot with a `NIL` value. When the ancestor view is opened (in response to an `Open` message), the view system creates the view for the child and sets the parent's slot value to point to that child. The slot is reset to `NIL` when the ancestor is closed, but as long as the ancestor is open, the declared view exists.

Sibling Messages and Declaring

Declaring a template to an ancestor makes it available not only to the ancestor, but to any other descendants of that ancestor (via parent inheritance). This is the way for siblings to send messages to each other. Here is a general rule for declaring:

- If one view wants to send a message to a second view, the second view should be declared in some ancestor common to both.

Declaring and Linked Subviews

There is one quirk associated with linked subviews and declaring. NTK provides no way to declare a child to an ancestor outside of its layout window. You will commonly need to declare the top most template in a layout (for instance, your detail or overview template) to the application base template. Since you can't

declare the topmost view in a linked layout, you declare the linking mechanism instead. Declare the LinkedSubView template in the main template. You also need to name the LinkedSubView (we suggest you give it the same name as the linked subview).

Invisible Views

If you have made a template invisible (in the `viewFlags` slot), you must declare that template to some ancestor. Otherwise, the view will never be created and you will have no access to it. Without access, you cannot send an `Open` message to an invisible view.

This will commonly be needed for slips that you open in certain circumstances or for other views that you open dynamically (rather than when the application opens). If your template is declared to an ancestor, the view will be created for the declared invisible template when that ancestor opens. Then, at some later point, you can send the `Open` message to that invisible (but existent) view.

Don't Declare All Views

Avoid the temptation to declare all your views. If you declared all your views to the application base view, then upon opening your application, every single view would be created—an unnecessary memory hit to say the least. Instead, only declare views you need to access by name.

Application Base View

The application base view is automatically declared to its parent, the root view (the application symbol is used as the slot name). The base view is created when the application is installed and deleted when the application is removed. This has an important ramification:

- Slots in the application base view retain their value even when the application is closed.

Do not be tempted, however, to save important information in the base view for later use. This information will be lost if the Newton is reset. If you want to save data, you should store it in soups (see Chapter 8). In the base view, only store unessential information (like what view or item the user was last looking at, or the last location of a movable window).

Since the base view remains when the application is closed, make sure to `NIL` out base view slots that are not needed. For example, if you create an array when the application opens, make sure to `NIL` out the slot in the `viewQuitScript` of the base view. Otherwise, the unneeded array will exist until the application is removed.

InstallScript and RemoveScript

The `InstallScript` is a function that is part of your application package. It executes when your application is installed. You should be aware of the three different times your application is installed:

1. The user explicitly installs the application using either Newton Connection or NTK or through Receive Enhancement in the Inbox.
2. The Newton gets reset (by either the user, the system, or an act of God). This causes each application to be reinstalled.
3. The user inserts a card. Each application on the card is installed.

The `RemoveScript` is called when your application is deinstalled. There are also two different times when it can be called:

1. The user explicitly deinstalls the application: using Remove Software in Preferences or Remove Software in Card.
2. The user removes a card. Each application on the card is deinstalled.

If the user removes the card, the `RemoveScript` is called after the card has been removed. Thus, the application's templates and protos are no longer available.

Both functions are created in your "Project Data" file (see "The Project Data File" on page 377). Each takes a `packageFrame` parameter, which contains information about your package, including the application symbol. Your `InstallScript` can add slots to the package frame that can then be retrieved in your `RemoveScript`.

Sample InstallScript and RemoveScript

These scripts are commonly used to register your application for certain system services (like Find and Intelligent Assistance). Here is an example:

```
InstallScript := func(package)
begin
// To be notified of changes to the soup
// (including a changed folder)
    AddArraySlot(soupNotify, kSoupName);
    AddArraySlot(soupNotify, kAppSymbol);

    // for Find All
    AddArraySlot(findApps, kAppSymbol);

    // for Intelligent Assistance
    packageFrame.taskTemplateID :=
      RegTaskTemplate(
      package.theForm.taskTemplate);
end;

RemoveScript := func(package)
begin
// remove soup name and app symbol
// from soupNotify array
    local soupNotifyPos := ArrayPos(
      soupNotify, kAppSymbol, 0, nil);
    ArrayRemoveCount(
      soupNotify, soupNotifyPos - 1, 2);

    // for Find All
    SetRemove(findApps, kAppSymbol);

    // for IA
    if package.taskTemplateID then begin
        UnRegTaskTemplate(package.taskTemplateID);
        package.taskTemplateID := nil;
    end;
end;
```

Warnings for RemoveScripts

Your `RemoveScript` should not try to access the templates or user protos of your application (remember, they are on the card that was just removed). If it attempts to do so, the user will get the dreaded "The card is still needed" slip. The application base view is still present when the `RemoveScript` is called, but the `_proto` slot of the base view still points at the (now-nonexistent) template; so don't try to use the base view either.

Keep RemoveScript Small

The `InstallScript` and `RemoveScript` are copied into the internal memory of the Newton when the application is installed. Once the `InstallScript` has executed, it is removed from memory. The `RemoveScript` remains in memory so that it will be available even if the card gets pulled. Since there is a limited amount of memory, you should keep your `RemoveScript` as small as possible.

Adding Code to WaiterHelper

We'll add code in small pieces to the WaiterHelper application.

Small Changes First

To begin with, create a slot named `menu` in your application base view (see "Adding Slots" on page 372). Paste in the frame you created the `menu` object you created in "Writing Code for WaiterHelper" on page 143 (see Figure 7.5).

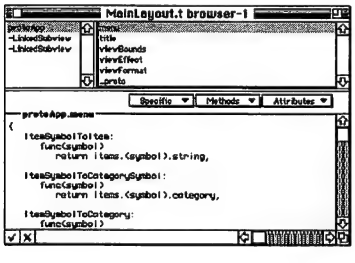


Figure 7.5 The `menu` slot in the application base view.

Declaring Some Templates

Since the "detail" template will be sending messages to many of its descendants, these descendants will need to be declared to the "detail" template. Likewise, the pickers need to be declared to "itemDetail":

1. Declare date, numPeople, tablePicker, table, order, title, and itemDetail to "Detail".

2. Similarly, declare commentPicker, itemPicker, categoryPicker to "itemDetail".
3. Make sure the detail template is visible, and the overview template is not visible (in the `viewFlags` slot).

Updating categoryPicker and itemPicker

Enabling categoryPicker

To begin with, let's have the categoryPicker get the categories from the menu object. Only the menu object needs to change if categories are renamed, added, or deleted. Add a `viewSetupFormScript` slot to categoryPicker:

```
func()
begin
   self.labelCommands := menu:GetCategories();
   inherited:?viewSetupFormScript();
end
```

Now, if you build and download, the category picker displays the categories from the menu object.

Enabling itemPicker

Next, let's add some code to update the itemPicker when the user chooses a category. The `protoLabelPicker` sends itself the `textChanged` message when the picker item changes. Add a `textChanged` slot to categoryPicker:

```
func()
begin
   SetValue(itemPicker, 'labelCommands,
      menu:CategoryToItems(entryLine.text));
   SetValue(itemPicker, 'text, entryLine.text);
end
```

This updates the items in the picker, as well as the label for the picker. The itemPicker still has to respond to those changes. Add a `viewChangedScript` slot to the itemPicker:

```
func(slot, view)
begin
   inherited:?viewChangedScript(slot, view);
   if slot = 'labelCommands then begin
     if not ArrayPos(labelCommands, entryLine.text,
         0, GetGlobals().functions.StrEqual) then
         :UpdateText(labelCommands[0]);
   end else if slot = 'text then
      :SyncView();   // update the label
end
```

This method looks for changes (due to `SetValue`) in slots in the view. If the `labelCommands` slot changes, the method determines whether the current chosen item is in the `labelCommands` array. If it is not, it changes the current chosen item (arbitrarily) to the first string in the array. If the text slot changes (the label in the `protoLabelPicker`), the method calls `SyncView`. This recreates the `viewCObject`, causing it to reread the text slot and redraw the new label.

At this point, when your run your application, choosing a new category updates the itemPicker (both the items and the label). Since the label changes width dynamically, however, we need to reserve enough room for wide labels. While we're at it, we can adjust the category, item, and comment pickers so that they all line up with one another. To do so:

- add an `indent` slot to the categoryPicker, itemPicker, and commentPicker with the value of `70`.

Using the numPeople Picker

The number of chairs around the table should always reflect the number of people in the party. To ensure this, we will modify the numPeople picker and the table template.

Adding a currentNumber

Add a `currentNumber` slot to numPeople. This will hold the current number of people displayed in the picker. Add a `textChanged` slot to numPeople. This will be called when the user chooses a new number:

```
func()
begin
   local num := StringToNumber(entryLine.text);
   // convert from real to integer
   if num then
      num := RIntToL(num);
   if num <> currentNumber then begin
      self.currentNumber := num;
      table:SetNumberPeople(num);
   end;
end
```

The code converts the picked string to a number and then (if it is different) sends the table the `SetNumberPeople` message.

Setting up the Chair Children

The table needs to iterate over its chair children. To begin with, add a `chairChildren` slot to the table template which will contain an array of chair views. This can not be initialized until run-time (views do not exist at compile time). Add a `viewSetupDoneScript` to the table which will initialize that array:

```
func()
begin
   self.chairChildren := :ChildViewFrames();
   // remove the tablePicker--the first child
   ArrayRemoveCount(chairChildren, 0, 1);
end
```

The code removes the one non-chair child from the array of views returned by `ChildViewFrames`. Finally, add the `SetNumberPeople` slot to the table template:

```
func(numberPeople)
begin
   // Show children 1..numberPeople-1
   // Hide children numberPeople..
   // If a child is already correct, leave it alone
   foreach i, child in chairChildren do begin
      if i < numberPeople and
          not Visible(child) then
         child:Show();
      else if i >= numberPeople and
          Visible(child) then
         child:Hide();
   end;
end
```

This method hides and shows the appropriate children. Now when you run the application, the correct number of chairs displays as you change the numPeople picker.

Displaying Bills in the Detail

Representing a Bill

To represent a bill, we'll use a frame. This is the structure (the slot names and what they will do):

`tableNumber`	Table number (integer).
`date`	Date and time (number of minutes since January 1, 1904).
`numberInParty`	Number of people in the party (between 1 and 4).
`checkNumber`	Check number (integer).
`orders`	Array of each person's order.

An individual order is represented as an array of items. An item in this array is a frame with these slots:

`itemSymbol`	A symbol representing the item (for example, `'coffee`, or `'|Roast Beast|`.
`comment`	Comment about this item (string).

In addition, we'll provide functions that operate in a separate object:

`SetNumberInParty(`*`bill`*`,`*`num`*`)`	Sets the number in party for *`bill`*. It adjusts the size of the orders array to match this number.
`GetTotal(`*`bill`*`)`	Returns the total amount of *`bill`* (as a real).

Note: It would make more sense to put the `SetNumberInParty` and `GetTotal` methods in a frame and have each bill proto from it. When we start saving bill frames in soups, however, we would lose the `_proto` slot. Therefore, we just put the functions in a separate object.

Creating a Random Bill

Add a `CreateRandomBill` method to the base view:

```
func(numPeople)
begin
   constant kMinutesPerDay := 1440;
   local randomOrders := [
      [{ itemSymbol: 'coffee},
       { itemSymbol: 'pie, comment: "Extra Hot"},
       { itemSymbol: 'spicyFries}],
      [{ itemSymbol: 'rootBeer},
       { itemSymbol: 'greekPizza,
         comment: "No anchovies"},
       { itemSymbol: 'bakedApple},
       { itemSymbol: 'spicyFries}],
      [{ itemSymbol: 'hotTea},
       { itemSymbol: 'vegetablePie},
       { itemSymbol: 'mashedPotatoes}],
   ];

   local bill := {
      numberInParty: 0,   // set by SetNumberInParty
       checkNumber: Random(101, 999),
          // random time in the last 24 hours
       date: Time() + Random(-kMinutesPerDay, 0),
       tableNumber: Random(51, 56),
       orders:[],
   };
   billFunctions:SetNumberInParty(bill, numPeople);

   for i := 0 to numPeople - 1 do
      bill.orders[i] :=
         DeepClone(randomOrders[Random(0,
        Length(randomOrders)-1)]);

   return bill;
end
```

Add a `billFunctions` slot to the base view as well:

```
{
   SetNumberInParty:
      func(bill, n)
      begin
         if n <> bill.numberInParty then
            SetLength(bill.orders, n);
         for i := bill.numberInParty to n-1 do
            bill.orders[i] := [{itemSymbol: 'none}];
         bill.numberInParty := n;
      end,
   GetTotal:
      func(bill)
      begin
         local total := 0.0;
         local menu := GetRoot().(kAppSymbol).menu;

         foreach order in bill.orders do
            foreach item in order do
               total := total +
                  menu:ItemSymbolToPrice(
               item.itemSymbol);
         return total;
      end,
}
```

Create a "Project Data" file (see "The Project Data File" on page 377) and add the following line:

```
constant kAppSymbol := '|ChangeMe:SIG|;
constant kApplicationName := "WaiterHelper";
constant kHiliteNow := true;
```

Displaying a Random Bill

Now we are ready to create a random bill (temporarily) and display it. Write a `viewSetupDoneScript` in the detail template:

```
func()
begin
   :DisplayBill(:CreateRandomBill(3));
end
```

Now write the `DisplayBill` method in detail:

```
func(bill)
begin
   // make bill accessible to children
   self.currentBill := bill;

   SetValue(title, 'text, "Check #" &
      NumberStr(bill.checkNumber));
   SetValue(date, 'text, DateNTime(bill.date));
   numPeople:UpdateText(
      NumberStr(bill.numberInParty));
   tablePicker:UpdateText(
      NumberStr(bill.tableNumber));
end
```

The method is not complete as it does not display the items. It will allow us to see most of the information in the bill, however. `UpdateText` is a `protoLabelPicker` method that updates the text in the picker.

Add a `currentBill` slot to the detail template. This slot is updated in `DisplayBill` to whatever bill is currently being displayed. Other views in the hierarchy can now have access to the current bill.

At this point, you can build and download, and you should see the correct number of tables, date, table number, and check number.

Displaying the First Person's Order

Add the following line to the end of the `DisplayBill` method of the detail template:

```
order:DisplayOrder(bill.orders[0], not kHiliteNow);
```

This will display the order of the first person.

Add a `currentOrder` slot to the order template. In addition, add `DisplayOrder`:

```
func(order, hiliteNow)
begin
   self.currentOrder := order;

   // remove any existing children
   if self.stepChildren then
       foreach child in :ChildViewFrames() do
         RemoveStepView(self, child);
   self.stepChildren := [];

   // create one view for each item in the order
```

```
        foreach item in order do begin
           local newView := AddStepView(self,
              pt_itemRow);
           newView:DisplayItem(item);
        end;

        if Length(order) > 0 then
           :DisplayOneItem(:ChildViewFrames()[0],
              hiliteNow);
     end
```

The code creates a `pt_itemRow` view for each item in the order. In addition, it displays the first item in the order in the itemDetail view. Since it adds views dynamically, you should remove the item1 and item2 children of order.

Add a `selectedItem` slot to order which represents the selected item. Add a `DisplayOneItem` method to order:

```
     func(itemRow, hiliteNow)
     begin
        if hiliteNow then
           itemRow:HiliteUnique(true);
        else // hilites real soon, but not immediately
           AddDeferredAction(
              func() itemRow:HiliteUnique(true), []);
        self.selectedItem := itemRow.currentItem;
        itemDetail:DisplayItem(selectedItem);
     end
```

Showing the First Order in itemDetail

Add a `currentItem` slot to the itemDetail template and also add a `DisplayItem` method:

```
func(item)
begin
   local string;

   self.currentItem := item;

   categoryPicker:UpdateText(menu:ItemSymbolToCategory(item.itemSymbol));
   itemPicker:UpdateText(menu:ItemSymbolToItem(item.itemSymbol));

   if item.comment then
      string := Clone(item.comment);
   else
      string := "";
         SetValue(commentPicker.entryLine, 'text, string);
end
```

This sets the text of the category, item, and comment views to their appropriate values.

Fixing the itemRow Proto

In addition, add a `currentItem` slot and `DisplayItem` method to the itemRow proto:

```
func(item)
begin
   self.currentItem := item;

   local string :=
      menu:ItemSymbolToItem(item.itemSymbol);

   if item.comment and StrLen(item.comment) > 0 then
      string := string && "(" & item.comment & ")";

   SetValue(self, 'text, string);
end
```

Run the application, and the items in the first order will display in each row.

Reflecting Picker Changes in itemRow

Now we have to get changes in the item or comment to change the item row. Add a `textChanged` method to the itemPicker:

```
func()
begin
   currentItem.itemSymbol :=
      menu:ItemToItemSymbol(entryLine.text);
   order:ItemChanged(currentItem);
end
```

Next, add an `ItemChanged` method to the order template:

```
func(item)
begin
   // redisplay item

   local position :=
      ArrayPos(currentOrder, item, 0, nil);
```

```
if position then begin
      local child := :ChildViewFrames()[position];
      child:DisplayItem(item);
   end;
end
```

Also, add a `textChanged` method to the commentPicker:

```
func()
begin
   local newComment;
   local hasChanged := nil;

   if entryLine.text = nil or
      StrLen(entryLine.text) = 0 then
      newComment := nil;
   else
      newComment := entryLine.text;

   // check for one NIL, and one non-NIL
   hasChanged :=
      (not newComment) <> (not currentItem.comment);

   if not hasChanged and newComment and
      currentItem.comment then
      hasChanged := not
         StrEqual(newComment, currentItem.comment);

   if hasChanged then begin
      currentItem.comment := Clone(newComment);
      order:ItemChanged(currentItem);
   end;
end
```

At this point, if you build and download, using the three pickers will update the display in the first in the row of items.

Enabling Tapping on an Item

Now, let's allow tapping on an item in the order view. Modify the itemRow proto to add a `viewFlags` slot and turn on `vClickable`. Add a `viewClickScript` to the proto:

```
func(unit)
begin
   InkOff(unit);   // turn off ink
   // hilite this row, unhilite any other rows
   :HiliteUnique(true);
   order:DisplayOneItem(self, kHiliteNow);

   return true;   // we handled the click
end
```

Build and download the application. You should now be able to click on different items to edit them.

New and Delete Buttons

Now, let's support the new and delete buttons.

New Button

Modify the `buttonClickScript` of the newItem button:

```
func()
begin
   order:AddNewItem();
end
```

Add an `AddNewItem` method to order:

```
func()
begin
   if Length(currentOrder) < 5 then begin
      newItem := {itemSymbol: 'none};
      AddArraySlot(currentOrder, newItem);
      newView := AddStepView(self, pt_itemRow);
      newView:Displayitem(newItem);

      :DisplayOneItem(newView, kHiliteNow);
    end else
      :Notify(kNotifyAlert,
         EnsureInternal(kApplicationName),
         EnsureInternal("Can't add more items."));
end
```

This limits the number of items for a given person to 5 (the amount that will fit in the view).

Delete Button

To support deleting an item, modify the `buttonClickScript` of the deleteItem button:

```
func()
begin
   order:DeleteSelectedItem();
end
```

Next, add a `DeleteSelectedItem` method to order:

```
func()
begin
   if Length(currentOrder) > 1 then begin
      SetRemove(currentOrder, selectedItem);
      // recreate views
      :DisplayOrder(currentOrder, kHiliteNow);
   end else
      :Notify(kNotifyAlert,
         EnsureInternal(kApplicationName),
         EnsureInternal(
         "Can't delete the last item."));
end
```

At this point, if you build and download, you can add and delete items from an order.

Displaying a Chair Person's Order

We need to display the order for a particular person when we click on the chair. To do so, we must modify the chair proto. Edit the `viewFlags` of chair, turning on `vClickable`. In addition, add the following `viewClickScript`:

```
func(unit)
begin
   InkOff(unit);
   :Parent():SelectChair(self, kHiliteNow);

   return true; // we handled the click
end
```

Add a `SelectChair` method to the table template:

```
func(chair, hiliteNow)
begin
   local chairNumber;

   chairNumber :=
      ArrayPos(chairChildren, chair, 0, nil);
   :SelectChairNumber(chairNumber, hiliteNow);
end;
```

Add a `selectedChairNumber` slot and a `SelectChairNumber` method to the table template:

```
func(chairNumber, hiliteNow)
begin
   local theChild := chairChildren[chairNumber];
   self.selectedChairNumber := chairNumber;

   if hiliteNow then
      theChild:HiliteUnique(true);
   else begin
      // hilite soon, but not immediately
      AddDeferredAction(
         func() theChild:HiliteUnique(true), []);
   end;
   order:DisplayOrder(
      currentBill.orders[chairNumber], hiliteNow);
end
```

Finally, modify `DisplayBill` in the detail template. Remove the line:

```
order:DisplayOrder(bill.orders[0], not kHiliteNow);
```

Replace it with:

```
table:SelectChairNumber(0, not kHiliteNow);
```

This way, the first chair will be highlighted to begin with. At this point, you can build and download; click on chairs to display different orders.

When Someone Leaves the Party

Modify `textChanged` in the numPeople template to:

```
func()
begin
   local num := StringToNumber(entryLine.text);
   // convert from real to integer
   if num then
      num := RIntToL(num);
   if num <> currentNumber then begin
      self.currentNumber := num;
      billFunctions:SetNumberInParty(
         currentBill, num);
      table:SetNumberPeople(num);
   end;
end
```

This makes sure that the order is updated when the number of people in the party changes.

Highlighting Chairs

To make we don't highlight a non-existent chair, modify the `SetNumberPeople` method of table to:

```
func(numberPeople)
begin
   // Show children 1..numberPeople-1
   // Hide children numberPeople..
   // If a child is already correct, leave it alone
   foreach i, child in chairChildren do begin
      if i < numberPeople and
         not Visible(child) then
         child:Show();
      else if i >= numberPeople and
         Visible(child) then
         child:Hide();
   end;
   if not selectedChairNumber or
      selectedChairNumber >= numberPeople then
      :SelectChairNumber(0, not kHiliteNow);
end
```

Now, you can build and download and change the number of tables from the picker; the chairs will update correctly.

Summary

In this chapter, you learned how the Newton view system creates the views in your application. You also found out about the components of a view and the three important messages the view system sends to each view when it is opened:

- `viewSetupFormScript`
- `viewSetupChildrenScript`
- `viewSetupDoneScript`

After detailing the other view methods you can use, we discussed declaring views as a way to give an ancestor access to a child's slots. Lastly, you saw the various views of WaiterHelper come into being.

Chapter 8

Newton Data Storage

Worries go down better with soup than without.

—*Yiddish proverb*

The Newton has a unique model of data storage and manipulation. Data is transparent to all applications and equally accessible regardless of its creator. The Newton user experiences this as a seamless integration of information across applications. No more entering data multiple times for multiple applications.

Introduction

This section provides you with a detailed overview of the Newton's model for data storage and retrieval. First, we describe the Newton model as it differs from traditional models, then we cover some of the central concepts of this model, including:

- Stores—the physical location of data.
- Soups—a collection of related entries.
- Queries and cursors—the way you order and navigate through data.

Persistent Objects

On most platforms, applications save data in specialized application and file formats (see Figure 8.1). Object-oriented applications that inhabit this world must convert their data objects to some arbitrary flat file format at write time and convert back at read time.

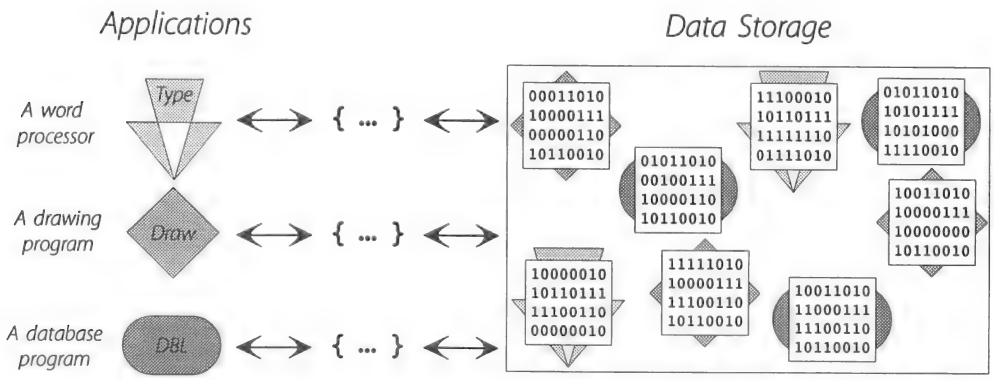


Figure 8.1 A traditional model of data file types and storage.

Newton provides a different type of data world. Rather than writing files, you store objects. Since you are storing objects, they are accessible to everybody without any conversion. On Newton, the data object is a frame. Every application can store frames, retrieve them, and modify frames they created or ones they did not (see Figure 8.2). In addition, searching for particular information is a straightfor-

ward process. Any application can search for a given frame or a set of frames that satisfy some criteria.

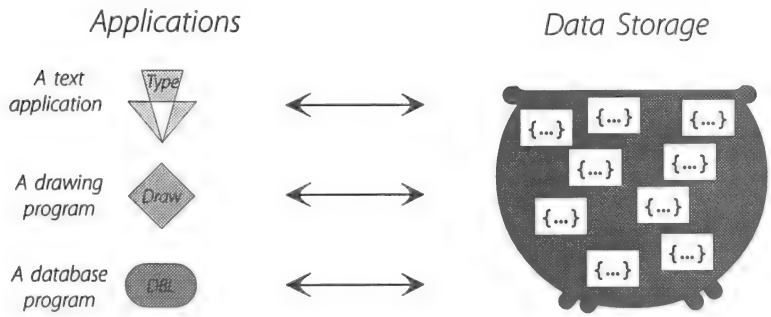


Figure 8.2 The Newton model of data and storage.

Within this data model, what remains consistent is the structure of the data object. Thus, we use the phrase *persistent object* to refer to these objects. They persist while stored and do not require explicit code to write them to or retrieve them from a file. On Newton, these persistent objects are called *entries*.

There are two advantages to storing objects directly:

- It is easier to write applications.
- Sharing data between applications is simpler.

On traditional platforms, applications have unique file formats that make sharing data a complex issue. On Newton, all storage is done using frames. Sharing data is as easy as retrieving an object and accessing its slots.

Soups

Here are a few of the important rules of Newton data objects:

- Related data entries are stored in a *soup*. For example, each name card in the Names application is an entry in a soup.
- Each soup has a unique name (a string). For example, the Names application stores its entries in a soup with the name "Names".
- A soup can contain one or more *indexes*.

Soup Indexes

A soup index is based on the value of a particular slot and allows quick lookup for entries based on that slot value. Indexes also provide sorting capabilities. For example, if a soup is indexed using a `name` slot that contains a text string, entries can be accessed in a sorted (alphabetical) order on that string.

These indexes are implemented with a tree based on slot values, similar to a binary tree, where the leaves of the binary tree point to actual entries (see Figure 8.3).

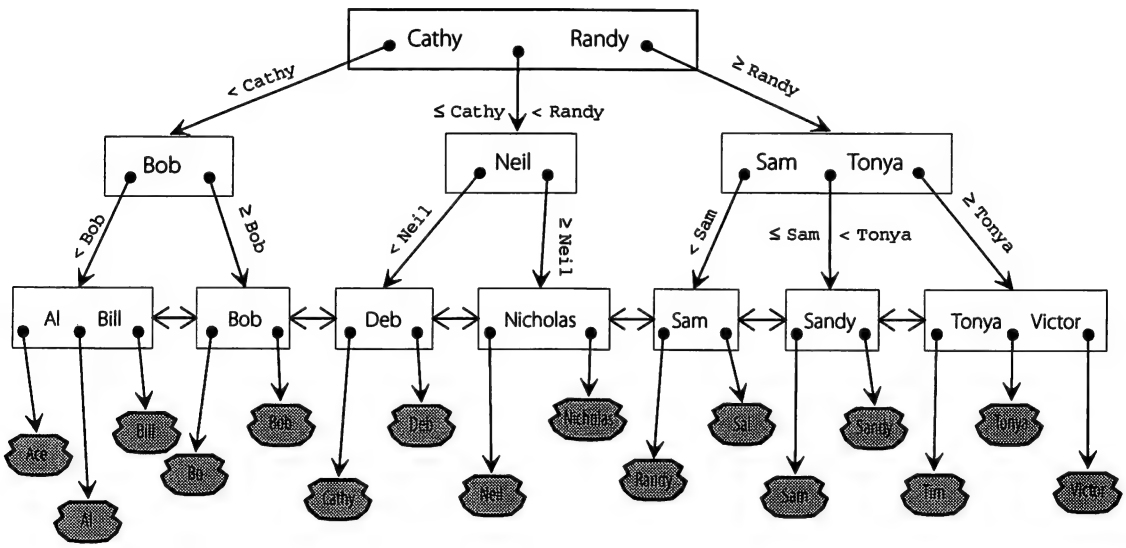


Figure 8.3 An index is a tree allowing quick access to a particular entry based on a key value.

Sharing Data between Applications

On desktop computers people have to enter identical data innumerable times. Each application requires that the data be put in its own format: you enter names, phone numbers and fax numbers in a computerized address book only to have to reenter the same information in your fax application when you want to send a fax. Copy, paste, importing and exporting, while a solution, are not the best; it would be better if your fax application used the same information your address book uses.

With the Newton, users enter information only once. When you send a fax, your Newton fax application get the name and fax number from the "Names" soup. To facilitate this model of sharing data, Newton applications can:

- read entries from any soup
- modify entries in any soup
- add new entries to any soup
- add new slots to entries in any soup

Reading Data from Another Soup

To read data from a soup, your application needs to know two things:

- the name of the soup
- the structure of an entry: the entry's slot names and values

The format of the soups used by the built-in applications is published by Apple. You are encouraged to publish the format of your soups so that they can be used by other applications.

Adding Slots to Other Soup Entries

Adding new slots to entries in other soups is also straightforward. For example, you might add a `favoriteNumber` slot to entries in the Names application, for use by your application. The Names application won't display this new slot's information, of course, but it will maintain the slot in entries. In other words, if your application sets the favorite number of "Joe Jones" to `3`, and then the user edits the name of "Joe Jones" to "Joe X. Jones" in Names, the `favoriteNumber` slot will still be maintained as `3`. The Names application just ignores new slots.

One Copy of Data

You are encouraged to have your applications read data from existing soups. For example, the name, address, and phone number of the Newton user are stored in a soup entry in the "System" soup. Any time you need this information, read it from the soup; don't force the user to reenter the information.

Stores

Newton data is stored in stores. A *store* is a physical device that stores data and is similar to hard drives, or volumes, on desktop machines. Each Newton has an internal store. By inserting a PCMCIA ROM or RAM card, it has an external store as well. As the time this book was written, Newtons only had these two stores. There may well be more stores in the future, however, as it is easy to imagine a Newton with more than one PCMCIA slot. You might also expect to see databases on a host machine appearing as soups in a pseudo-store.

In your understanding of stores, there are two important points to remember:

- You may assume that the first store is the internal store, but nothing else. The second store may or may not be the external store, and the number of stores is unlimited.
- Stores may appear or disappear while an application is running—the user can eject or insert a PCMCIA card at any time.

Union Soups

As the name implies, a union soup is one larger soup combined from similar soups across multiple stores (see Figure 8.4). For example, a "Names" union soup would include the entries from all the "Names" soups on all resident stores. Here are the important points about union soups:

- Union soups are dynamic. As stores appear or disappear, the union soup will flex its size to match—it contains more or fewer entries.
- You do not normally care what store an entry is on, only what soup it is a member of. As a result, your program will almost always use union soups rather than soups on a particular store.
- When a soup is modified, you can get notified—normally when a card is inserted or deleted. When this happens, you'll probably want to redisplay your view with the current entries from the soup.

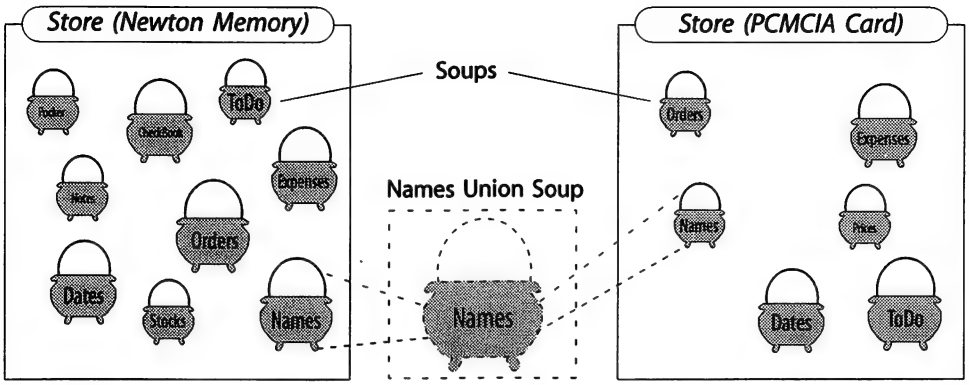


Figure 8.4 A "Names" union soup.

Queries/Cursors

A *cursor* is an object that iterates over entries in a soup. Cursors can:

- retrieve the current entry
- move forwards and backwards
- do searches

You create a cursor by making a query. A *query* specifies an index, which controls the order the cursor iterates over entries. Within a query, you can also restrict by particular conditions, specific text, or by index values that match a particular key. Using restricted queries, here are some examples of the way you can iterate a cursor:

- to entries that contain the word "Joe" in some slot
- through the entries based on the value in an indexed slot
- entries that have an even number of slots
- entries whose `name` slot starts with 'K'

Cursor are completely dynamic. For instance, if you've created a cursor that iterates over names beginning with 'K' and insert a card, the cursor will now also include those names on the card beginning with a 'K'. These means the next item the cursor iterates to may change, as shown in Figure 8.5.

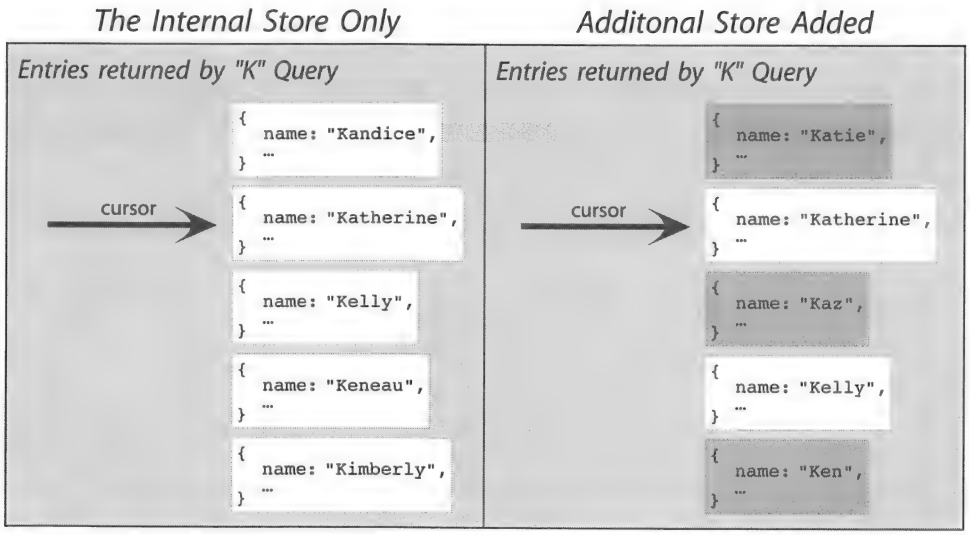


Figure 8.5 A dynamic cursor iterating over K entries.

Soup Entries Are Self-contained

You can add any frame to a soup. When you add a frame to a soup, nothing in the frame points outside the entry. Any slots in the soup frame containing references (frames, arrays) are cloned.

The cloning occurs recursively to ensure that all references are properly copied. The most straightforward consequence of this is that you need to exercise care with what type of frame you add to a soup. You should also consider the converse situation: retrieving an entry means you are getting the entry's entire structure (including the cloned slots).

Consider the foolishness of adding a view frame to a soup. The view would be added. Next, the cloning follows the view's parent chain, merrily duplicating its parent and all its ancestors right up to and including the root view. Clearly, the store wouldn't be large enough to hold such an entry. An entry, then, is not just a frame, but everything the frame points to.

Note: The `_proto` slot of a frame receives special treatment. The `_proto` slot is stripped from a frame before adding it to a soup. The theory behind this is that the `_proto` slot points to ROM or pseudo-ROM, and therefore doesn't need to be saved in a soup.

Soup Entries Have a Unique ID

When you add a frame to a soup, a new slot, `_uniqueID`, is added to the frame. This slot contains a number that is unique across all entries in the soup on that store. Note that two entries in the same union soup can have the same unique ID if they are on different stores.

Entries in a soup also have a `_modTime` slot if they have ever been modified. The slot contains the date they were modified (as always, in minutes since January 1, 1904).

Soup Entry Compression

Soup entries are compressed within the soup. Thus, an entry usually takes less space in the soup than it does as a frame in the frame heap. Since an entry is compressed, it isn't possible to determine ahead of time whether there will be enough room to add it. Instead, you must attempt to add an entry; the operation will fail if there is not enough room on the store.

Fortunately, iterating over entries with a cursor does not require a full retrieval of the entry. A dehydrated version is used to speed up the process. It is only when you try to read from or write to an entry that the entry is fully hydrated.

Description of Methods and Functions

Now that you have had a general introduction to data storage and are familiar with the important terms, we can describe the methods and functions you will use with soups. (Some rarely used functions are not covered here.) We will go from the outside in, by covering store methods first, then soups, then entry methods, and finally cursor and query functions.

Store Functions

While most applications don't use these store functions, we discuss them here for your reference.

GetStores

`GetStores()`

This global function returns an array of stores. The first entry in the array is the internal store.

```
Print(GetStores()); // without a card inserted
[{_Parent: {#3B6531},
  _proto: {#4403961},
  Store: 17853709,
  soups: [#440D3A1]}]

Print(GetStores()); // with a card inserted
[{_Parent: {#3B6531},
  _proto: {#4403961},
  Store: 17853709,
  soups: [#440BD09]},
 {_Parent: {#3B6531},
  _proto: {#440C419},
  Store: 17862951,
  soups: [#440C459]}]
```

Soup and Index Creation

`store:CreateSoup(soupName, arrayOfIndexes)`

This store method creates a soup named *soupName* on *store*. The second parameter is an array of frames. Each frame in that array specifies a soup index and has this structure:

```
{structure: 'slot, path: 'pathExpr, type: 'dataType}
```

Here is a description of each of the three slots:

`structure` The kind of index. Currently, you can only index on the value of a slot in an entry frame—thus this slot value is always: `'slot`.

`path` The path of the indexed slot. It can be a symbol like `'name` or `'date`, or a path expression like `'name.first`.

`type` This specifies the type of value contained in the indexed slot. Each entry must contain the same type of value in the indexed slot. For example, if you index on a name slot, you have a string value in this slot. The possible values for type are `'int`, `'string`, `'char`, `'real`, or `'symbol`.

If you pass an empty array, no index will be created. Here is an example that creates a soup with two indexes:

```
internal := GetStores()[0];
internal:CreateSoup("Foo", [
   {
      structure:  'slot,
      path:       'name,          — Index #1
      type:       'string
   },
   {
      structure:  'slot,
      path:       'height,        — Index #2
      type:       'real
   }
]);
```

Note: Normally, you will use `RegisterCardSoup` in your application. It creates soups on all stores and is the recommended way to create soups. See "Creating Your Soup" on page 239 for complete details.

Getting Soup Names

---

store:GetSoup(*soupName*)

This method gets the soup named *soupName* from *store*. If no such soup exists, the method returns NIL.

Here are two examples of how you would use it. The first example returns the "Names" soup on the Newton and the second returns NIL because no soup exists:

```
internal := GetStores()[0];

aSoup := internal:GetSoup("Names");
Print(internal);
{_Parent: {#3B5FE9},
 _proto: {#4408E91},
 tStore: 17853709,
 storeObj: {#4403981},
 theName: "Names",
 cache: [#440B4D9],
 cursors: [#440B149]}

aSoup := internal:GetSoup("Dinglehopper");
Print(aSoup);
NIL
```

Finding Soups

---

store:HasSoup(*soupName*)

This method returns true if *store* contains a soup named *soupName*. It returns NIL otherwise. Here are two examples, one returning a soup, the other not:

```
internal := GetStores()[0];

Print(internal:HasSoup("Names"))
TRUE

Print(internal:HasSoup("Dinglehopper"))
NIL
```

---

store:GetSoupNames()

This method returns an array containing the names of all the soups on a specified *store*. Here is an example:

```
internal := GetStores()[0];
Print(internal:GetSoupNames())
["Calendar", "Calendar Notes", "Directory", "Inbox",
"Library", "Names", "Notes", "Outbox", "Repeat Meet-
ings", "Repeat Notes", "System", "To do"]
```

Miscellaneous Store Function

---

store:GetName()

Each store has a name; this method returns that name. Here are examples:

```
internalStore := GetStores()[0];
Print(internalStore:GetName());
"Internal"

externalStore := GetStores()[1];
Print(externalStore:GetName());
"Card"
```

---

store:SetName(*newName*)

Sets the name of *store* to *newName* (if the store is writable).

---

store:TotalSize()

Returns the total size of *store* in bytes. For example:

```
internalStore := GetStores()[0];
Print(internalStore:TotalSize());
156416
```

```
externalStore := GetStores()[1];
Print(externalStore:TotalSize());
1943552
```

---

store:UsedSize()

Returns the number of bytes currently used in *store*. Here are some examples:

```
internalStore := GetStores()[0];
Print(internalStore:UsedSize());
140220

externalStore := GetStores()[1];
Print(externalStore:UsedSize());
598728
```

---

store:SetSignature(*newSignature*)

When a store is created it has a random integer assigned to it. This signature integer provides a way to distinguish between stores that have the same name. SetSignature changes the signature of *store* to the integer *newSignature*. You shouldn't normally call this method.

---

store:GetSignature()

This method returns the signature of *store*. For example:

```
internalStore := GetStores()[0];
Print(internalStore:GetSignature())
-419184440

externalStore := GetStores()[1];
Print(externalStore:GetSignature())
219712247
```

---

store:GetKind()

This method returns the media type of *store*. The possibilities (for Newtons localized for the U.S.) are currently "Internal", "Flash storage card", "Storage card" and "Application card". Here are some examples:

```
internalStore := GetStores()[0];
Print(internalStore:GetKind())
"Internal"

externalStore := GetStores()[1];
Print(externalStore:GetKind())
"Flash storage card"
```

---

store:IsReadOnly()

This returns true if *store* is in a read-only state (for example, a ROM card or a write-protected RAM card). This returns NIL if *store* can be written.

---

store:Erase()

This completely erases *store*. This is not a call to make lightly—be very certain that the user wishes you to do this.

Soup Functions

There are as many of these functions as there are varieties of soup in the grocery store. Most applications use only the three most important methods. They create a union soup with GetUnionSoup, add entries to the correct store with AddToDefaultStore, and call BroadcastSoupChange to notify other applications about soup changes. There are many other soup functions that we discuss after these important three, but you won't use them often.

Creating a Union Soup

`GetUnionSoup(soupName)`

This function returns a union soup of all the soups named *soupName* from all available stores. As we said earlier, the union soup is dynamic; as stores are added or deleted, the size and entries in the union soup change. Here is the standard way you call this method to create a union soup:

```
allNames := GetUnionSoup("Names");
Print(allNames);
{_Parent: {#3B5C69},
 class: UnionSoup,
 soupList: [#440BF61],
 theName: "Names",
 cursors: [#440B099]}
```

Adding an Entry to the Union Soup

`unionSoup:AddToDefaultStore(frame)`

This method takes *frame* and adds it to *unionSoup*. The entry frame is placed on the default store. The user specifies the default store in the Card slip. An error will be generated if the default store doesn't contain that soup. Here is a standard use of the method:

```
theSoup:= GetUnionSoup("NameAndHeights");
theSoup:AddToDefaultStore(
   {name:"Neil", height: 72.5});
```

Soup Change Notification

`BroadcastSoupChange(soupName)`

When you change an entry in a soup, you should call the global function `BroadcastSoupChange`. It notifies other applications that are displaying entries from

the soup that the soup has changed. Adding or deleting stores also triggers a call to this function for each soup on the store in question.

Applications registered in the `soupNotify` global array are notified of changes to soups. To register, an application adds two entries to this array: the soup name and the application symbol. When `BroadcastSoupChange` is called for a particular soup, it marches through the array looking for matching soups. For every match, it sends the `soupChanged` message to the base view of the registered applications.

If your application needs to be notified for more than one soup, you simply register multiple times. The application symbol stays the same, while the soup name varies. See "Handling Changes to Your Soup" on page 243 for more information.

Miscellaneous Soup Methods

---

soup`:GetIndexes()`

This method returns an array of frames, one for each index in *soup*. This method is not supported for union soups. Here are some examples of its use:

```
internalSoup := GetStores()[0];

namesSoup := internalSoup:GetSoup("Names");
Print(namesSoup:GetIndexes());
[{structure: slot,
  path: sortOn,
  type: String,
  index: 125}]

calendarSoup := internalSoup:GetSoup("Calendar");
Print(calendarSoup:GetIndexes());
[{structure: slot,
  path: mtgStartDate,
  type: Int,
  index: 96},
 {structure: slot,
  path: mtgAlarm,
  type: Int,
  index: 98}]
```

```
soup:RemoveIndex(indexPath)
```

You use this to remove a `soup` index on the path *`indexPath`*. This method works on both union and nonunion soups.

```
soup:SetSignature(signature)
```

When a soup is created, it is assigned a random integer as a signature. With this, you can tell whether a soup has been removed and a new soup added with the same name. This method sets the signature of *`soup`* to *`newSignature`* and is not supported for union soups.

```
soup:GetSignature()
```

This method returns the signature of *`soup`* and is not supported for union soups.

```
soup:CopyEntries(destSoup)
```

This method copies all the entries from *`soup`* to *`destSoup`*. Neither *`soup`* nor *`destSoup`* may be a union soup.

```
soup:GetNextUid()
```

This returns the value of the `_uniqueID` that will be assigned to the next entry added to *`soup`*. This method is not supported for union soups.

Cursor Functions

There are some important functions that you will use to set up your soup queries and to control the movement of your cursors. First, let us look at how you create a query with and without an index.

```
Query(soup, queryFrame)
```

Calling `Query` creates a cursor for *`soup`* based on the criteria specified in *`queryFrame`*. The qualified query entries are then iterated over using the cursor. As we said before, both queries and soups are dynamic. Adding or deleting entries from the union soup can change the number and order of entries in the query.

Creating a QueryFrame

```
{type: 'value, … optionalSlots}
```

`queryFrame` is a frame that specifies the retrieval criteria for entries. It can also indicate the order in which to retrieve them. You do not, however, have to place restrictions on a query. Creating a query using this query frame will return every entry in the soup:

```
Query(unionsoup, {type: 'index})
```

Here is a list of the one required and six optional slots in `queryframe`:

`type`	*Required.* Possible values are: `'index` to use an index `'words` to search for words `'text` to search for text.
`indexPath`	*Optional.* Specifies the path of an indexed slot. Entries are sorted by the value of this slot.
`startKey`	*Optional for `index` queries.* The first entry returned has an indexed slot value greater than or equal to the value of `startKey`.
`endTest`	*Optional for `index` queries.* This function takes an entry and returns `true` or `NIL`. If the function returns a non-`NIL` value, this and all following entries are not part of the query. Otherwise, the entry is part of the query.
`words`	*Required for `word` queries.* This slot contains an array of strings. The query only accepts entries

with strings that match each of the array strings.

`text` *Required for* `text` *queries.* The value is a string. An entry will qualify for this query if it has a string containing `text`'s value in it. The text can be anywhere in the string.

`validTest` *Optional.* This function takes an entry and returns `true` or `NIL`. Each potential query entry is passed to this function. If it returns `NIL`, the entry is not part of the query, otherwise it is. Each time the cursor is moved, `validTest` is called.

Moving Cursors

There are a number of functions that let you move the cursor around. You can go forward or backwards, jump a certain amount, or go to a particular location.

---

cursor`:Entry()`

This returns the current entry from *cursor*. If the entry gets deleted while the cursor is still on it, `Entry` returns `'deleted` and no longer rests on a soup entry.

---

cursor`:Next()`

As the name implies, this moves *cursor* to the next entry satisfying the query and then returns it. If you reach the end of the line and there is no next entry, `Next` returns `NIL`.

---

cursor`:Prev()`

Straightforward as the last, this moves *cursor* backward to the previous entry that satisfies the query and returns it. If you are at the beginning and there is no previous entry, `Prev` returns `NIL`.

`cursor:Goto(entry)`

This function moves *cursor* to *entry*. The *entry* must be an entry in the soup and must have already been returned by a call to `Entry`, `Next`, or `Prev`.

`cursor:GotoKey(keyValue)`

This moves *cursor* to the first entry with an indexed value equal to *keyValue*. If no such entry is present, the cursor is put on the next indexed entry. This method should only be called for queries that have specified an `indexPath`.

`cursor:Reset()`

This moves *cursor* back to the first entry that satisfies the query.

`cursor:Move(numEntries)`

This moves *cursor* forward or backward by *numEntries*. `Move(1)` is equivalent to `Next`. `Move(-1)` is equivalent to `Prev()`. This function is faster than repeatedly calling `Next` or `Prev`.

`cursor:Clone()`

The `Clone` method returns a new cursor that shares the same query as *cursor*. The new cursor has an independent location, and as such provides you a way to have multiple iterators over the same entries.

Note: Don't use the global functions `Clone` or `DeepClone` on a cursor; use this `Clone` method instead.

`MapCursor(cursor, applyFunction)`

This function returns, in an array, a subset of the entries in the cursor. `MapCursor` calls *`applyFunction`* for each entry as a cursor iterates over it. *`applyFunction`* takes the entry as its parameter and returns a result. If `applyFunction` is itself `NIL`, then `MapCursor` returns the full set of `query` entries. Otherwise, `applyfunction` returns either `NIL` or the entry; if it returns the entry, it is appended to the `MapCursor` array.

Entry Functions

These are the global functions for entries. Applications often use only `EntryChange` and `EntryRemoveFromSoup`. Other functions are used less often.

`EntryChange(entry)`

This puts a modified entry back into the soup and updates the `_modtime` slot to reflect the current time. For example:

```
namesSoup := GetUnionSoup("Names");
s := Query(namesSoup, {type: 'index});

e := s:Entry();
Print(e);
```

A soup entry

```
{sortOn: "",
 cardType: 0,
 phones: [],
 email: NIL,
 company: NIL,
 address: NIL,
 address2: NIL,
 city: "Sherman Oaks",
 region: NIL,
 country: NIL,
 postal_code: NIL,
 bday: NIL,
 name: {first: "Joe",
        class: person},
 labels: nil,
 _uniqueID: 0,
 _modtime: 47234197}
```

We are going to modify this slot

```
e.address := "123 Main Street"; — Change slot value
EntryChange(e); ———————————— Update entry
Print(e);

{sortOn: "",
 cardType: 0,
 phones: [],
 email: NIL,
 company: NIL,
 address: "123 Main Street", ——— New slot updated in soup
 address2: NIL,
 city: "Sherman Oaks",
 region: NIL,
 country: NIL,
 postal_code: NIL,
 bday: NIL,
 name: {first: "Joe",
        class: person},
 labels: nil,
 _uniqueID: 0,
 _modtime: 47371192}
```

`EntryRemoveFromSoup(`*`entry`*`)`

This removes *entry* from its soup. For example:

```
namesSoup := GetUnionSoup("Names");
s := Query(namesSoup, {type: 'index});
e := s:Entry();
EntryRemoveFromSoup(e);
Print(s:Entry())
deleted
```

`EntryUndoChanges(`*`entry`*`)`

This routine rereads the original entry from its soup. Any changes made to *entry* since it was last read from or written to the soup are lost. For example:

```
namesSoup := GetUnionSoup("Names");
s := Query(namesSoup, {type: 'index});
e := s:Next();
Print(e);
```

```
{  sortOn: "LaGaly",
   region: NIL,
   name: {  first: "Carolyn",
            last: "LaGaly"},
   _uniqueID: 11,
   _modtime: 47130480}

e.region := "CA";
Print(e);
{  sortOn: "LaGaly",
   region: "CA",
   name: {  first: "Carolyn",
            last: "LaGaly"},
   _uniqueID: 11,
   _modtime: 47130480}

EntryUndoChanges(e);
Print(e);
{  sortOn: "LaGaly",
   region: NIL,
   name: {  first: "Carolyn",
            last: "LaGaly"},
   _uniqueID: 11,
   _modtime: 47130480}
```

`EntryReplace(`*`oldEntry, newFrame`*`)`

This copies *newFrame* into *oldEntry*, while retaining the `_uniqueID` of *oldEntry*. It updates the `_modTime` to the current time. This function updates the soup, making a call to `EntryChange` unnecessary. Here is an example:

```
s := GetUnionSoup("Names");
q := Query(s,
   {type: 'index, indexPath: 'sortOn});
e := q:Entry();
Print(e);
{  sortOn: "",
   address: NIL,
   _uniqueID: 15,
   _modtime: 47305764}

EntryReplace(e, {sortOn: "foo", address: "bar"});
Print(e);
{  sortOn: "foo",
   address: "bar",
   _modtime: 47372381,
   _uniqueID: 15}
```

`EntrySoup(`*`entry`*`)`

This function takes an entry and returns its soup. The soup returned is not a union soup but the one on the entry's actual store. Here is an example of its use.

```
printDepth:=-1;
internalStore:=GetStores()[0];
Print(internalStore:GetSoup("Names"));
{#440F7F1}

externalStore:=GetStores()[1];
Print(externalStore:GetSoup("Names"));
{#440F991}

s := GetUnionSoup("Names");
Print(s);
{#440F979}

q := Query(s, {type: 'index});

e := q:Entry();
Print(EntrySoup(e));
{#440F991}

e := q:Next();
Print(EntrySoup(e));
{#440F7F1}
```

External Store Address — `{#440F991}`

Internal Store Address — `{#440F7F1}`

`EntryStore(`*`entry`*`)`

Similar to the last function that returned an entry's particular soup, `EntryStore` returns an entry's store. For example:

```
printDepth:=0;
Print(GetStores());
[{#44039B1}, {#4409BA1}]

printDepth:=-1;
s := GetUnionSoup("Names");
q := Query(s, {type: 'index});

e := q:Entry();
Print(EntryStore(e));
{#4409BA1}
```

```
e := q:Next();
Print(EntryStore(e));
{#44039B1}
```

`EntrySize(entry)`

This returns the number of bytes *entry* occupies in the soup. This is the compressed size and not the size the entry will occupy in memory. Here is an example of its use:

```
s := GetUnionSoup("Names");
q := Query(s, {type: 'index});
e := q:Entry();
Print(e);
{  sortOn: "LaGaly",
   cardType: 0,
   company: NIL,
   address: NIL,
   name: {  first: "Carat",
            class: person,
            last: "LaGaly"},
   _uniqueID: 11,
   _modtime: 47371491}

Print(EntrySize(e));
82

e.address := "xyzzzzzzzzzzz2222";
EntryChange(e);
Print(EntrySize(e));
102
```

Compressed size of entry—

Entry's new size

`EntryTextSize(entry)`

This returns the byte size of `entry`'s text slots in the soup. Once again, this is a compressed size. Here is an example:

```
s := GetUnionSoup("Names");
q := Query(s, {type: 'index});
e := q:Entry();
Print(e);
```

```
{  sortOn: "LaGaly",
   cardType: 0,
   company: NIL,
   address: NIL,
   name: {  first: "Carat",
            class: person,
            last: "LaGaly"},
   _uniqueID: 11,
   _modtime: 47371491}

Print(EntryTextSize(e));
15

e.address := "1234567890";
EntryChange(e);
Print(EntryTextSize(e));
26
```

`EntryCopy(`*`entry, newSoup`*`)`

This function copies *entry* and adds the new entry to *newSoup*. If *newSoup* is a union soup, the copy will not necessarily be added to the default store. The function returns the new entry, leaving *entry* unchanged.

`EntryMove(`*`entry, newSoup`*`)`

This moves *entry* to *newSoup*, removing it from its existing soup. If *newSoup* is a union soup, the copy is not necessarily added to the default store. The function returns the modified entry (`_uniqueID` is modified). For example:

```
internalSoup := GetStores()[0]:GetSoup("Names");
externalSoup := GetStores()[1]:GetSoup("Names");
q := Query(internalSoup, {type: 'index, indexPath:
'sortOn});

e := q:GotoKey("LaGaly")
Print(e);
{  sortOn: "LaGaly",
   address: "1234567890",
   name: {  first: "Carat",
            class: person,
```

```
                    last: "LaGaly"},
            _uniqueID: 172,
            _modtime: 47372407}

        EntryMove(e, externalSoup);
        Print(e);
        {   sortOn: "LaGaly",
            address: "1234567890",
            name: {  first: "Carat",
                     class: person,
                     last: "LaGaly"},
            _uniqueID: 21,
            _modtime: 47372407}
```

Modified unique ID

`EntryModTime(entry)`

This returns the last modification time of *entry*. You should use this routine instead of directly accessing an entry's `_modtime` slot. In cases where *entry* has not been modified since it was originally placed in the soup, `EntryModTime` returns `0`. The time is an integer equivalent to the minutes passed since midnight, January 1, 1904.

`EntryUniqueID(entry)`

This returns the unique ID of *entry*. You should use this routine instead of directly accessing the `_uniqueID` slot of an entry.

`EntryChangeWithModTime(entry)`

This puts a modified *entry* back into the soup and updates the modification time.

`EntryReplaceWithModTime(entry, replacementEntry)`

This replaces *entry* with *replacementEntry* and updates the modification time.

`FrameDirty(entry)`

If `FrameDirty` thinks *`entry`* has been modified since it was last read from or written to the soup, it returns `true`. Otherwise it returns `NIL`. `FrameDirty` assumes modification to an `entry` if:

(1) the `entry`'s slots have been assigned to

(2) any frame's `entry` references have slots that have been assigned to

`FrameDirty` can be fooled. For instance, assigning to a character within a string does not return true. Watch this clever trick:

```
e := q:GotoKey("Foo");
Print(e);
{sortOn: "foo",
 address: "bar",
 _uniqueID: 15,
 _modtime: 47372454}

Print(FrameDirty(e));
NIL

e.address[1] := $x;          Replacing "bar" with "bxr" in address
Print(FrameDirty(e));
NIL

e.extra := {slot1: 11};      Adding a new slot to entry
Print(e)
{sortOn: "foo",
 address: "bxr",
 _uniqueID: 15,
 _modtime: 47372454,
 extra: {slot1: 11}}

Print(FrameDirty(e));
TRUE

EntryChange(e);
Print(FrameDirty(e));
NIL

e.extra.slot1 := 10
Print(FrameDirty(e));
TRUE
```

`FrameDirty` should return `true`

`FrameDirty` should return `true`

Samples in the Inspector

In this next section, we give you some sample code that uses these various soup and cursor functions. From these examples, you should be able to get a reasonable idea of how to implement the same methods in your own code. Here is what we implement:

- Set up procedures to print all the names in the "Names" soup in both unsorted and sorted order.
- Show you three different ways to print all the names beginning with "K" in the Names soup.
- Change an entry in an existing soup.
- Add slots to entries in an existing soup.
- Find the last entry in an index.
- Find all the names that match a string.
- Remove a store while a cursor is iterating.
- Show you what happens when you forget to call `EntryChange`.
- Index on multiple slots in an entry.

Printing All the Names in the Names Soup

Notice that in each example we get the union soup and create a cursor. We then get an entry and use a `while` loop to iterate through each one. We print the value of the `e.sortOn` slot and last of all move the cursor to the next entry.

In Unsorted Order

```
soup := GetUnionSoup("Names");
curs := Query(soup, {type: 'index);

e := curs:Entry();
while e do begin
   print(e.sortOn);
   e := curs:Next();
end;
```

In Sorted Order

```
soup := GetUnionSoup("Names");
curs := Query(soup,
   {type: 'index, indexPath: 'sortOn});

e := curs:Entry();
while e do begin
   print(e.sortOn);
   e := curs:Next();
end;
```

Printing All Names Beginning with 'K'

In the following three examples, we show you how to print out a particular set of names. Each subsequent example offers some refinements that make it more efficient than its predecessor.

Slow Way

The approach uses only a `validTest` to find names beginning with 'K'. The cursor must iterate through every single name in the soup, applying the `validTest` to each one. The example took approximately 9 seconds to execute.

```
soup := GetUnionSoup("Names");
curs := Query(
   soup,
   {
      type: 'index,
      indexPath: 'sortOn,
      validTest:
         func(e)
         begin
            return BeginsWith(e.sortOn, "k");
         end
   });

e := curs:Entry();
while e do begin
   print(e.sortOn);
   e := curs:Next();
end;
"Keohane"
"Kohnlenberger"
"Kollmyer"
"Kuang"
```

Faster Way

The names are indexed by a `sortOn` slot, so if we provide a `startKey` the cursor can go directly to the first name beginning with a 'K'. All the names after that are still visited, however, and the `validTest` is still applied to each. This example took approximately 6 seconds to execute:

```
soup := GetUnionSoup("Names");
curs := Query(
   soup,
   {
      type: 'index,
      indexPath: 'sortOn,
      startKey: "k",
      validTest:
         func(e)
         begin
            return BeginsWith(e.sortOn, "k");
         end
   });

e := curs:Entry();
while e do begin
   print(e.sortOn);
   e := curs:Next();
end;
"Keohane"
"Kohnlenberger"
"Kollmyer"
"Kuang"
```

Fastest Way

This `endTest` allows the cursor to start at the first name beginning with 'K', proceed through all the 'K' names, and end once it reaches the first non-'K' name. This example took approximately 2 seconds to execute, a big savings:

```
soup := GetUnionSoup("Names");
curs := Query(
   soup,
   {
      type: 'index,
      indexPath: 'sortOn,
      startKey: "k",
      endTest:
         func(e)
```

```
            begin
               return not BeginsWith(e.sortOn, "k")
            end
      });

   e := curs:Entry();
   while e do begin
      print(e.sortOn);
      e := curs:Next();
   end;
   "Keohane"
   "Kohnlenberger"
   "Kollmyer"
   "Kuang"
```

Changing an Entry in an Existing Soup

Here is an example of changing an entry in the Names soup. First we get a union soup for Names, create a cursor, placing it on an entry and printing it out in the Inspector.

```
namesSoup := GetUnionSoup ("Names");
namesCursor := query(namesSoup, {type: 'index});
anEntry := namesCursor:Entry();

Print(anEntry);
#440FC49  {sortOn: "Christie",
           cardType: 0,
           phones: [],
           email: NIL,
           company: NIL,
           address: NIL,
           address2: NIL,
           city: NIL,
           region: NIL,
           country: NIL,
           postal_code: NIL,
           bday: NIL,
           name: {first: "Agatha",
                  class: person,
                  last: "Christie"},
           labels: Personal,
           _uniqueID: 11,
           _modtime: 47130480}
```

Now, open the Names application and scroll to that entry. In the following code, we change some of the values in the entry and then send the `EntryChange` message. Last of all we send the `BroadcastSoupChange` message to update affected soups:

```
anEntry.city := "London";
anEntry.country := "England";
EntryChange(anEntry);
BroadcastSoupChange("Names");
```

You should see your entry updated on screen as well.

Adding Slots to Entries in an Existing Soup

Adding entries to an existing soup is straightforward. You use the same steps as those in the last exercise. You get a union soup, create a cursor, and put it on the entry. You add the slots to the entry by assignment. For example, an entry can get a new slot as easy as this:

```
anEntry.newSlot := "Dead Mystery Writer";
EntryChange(anEntry);
```

Now if you print the entry you can see the new slots:

```
#440FC49  {sortOn: "Christie",
   cardType: 0,
   phones: [],
   email: NIL,
   company: NIL,
   address: NIL,
   address2: NIL,
   city: NIL,
   region: NIL,
   country: NIL,
   postal_code: NIL,
   bday: NIL,
   name: {first: "Agatha",
                 class: person,
                 last: "Christie"},
   labels: Personal,
   _uniqueID: 11,
   _modtime: 47130480}
   newSlot: "Dead Mystery Writer"}
```

A new slot in the entry

The new slot will not be displayed in the Names application. Names will maintain the new slot for us, however. In our own program, we could display the full name entry with the extra slots by simply taking them into account in our display routines.

Finding the Last Entry in an Index

It is easy to get to the first entry in an indexed order: just create a cursor and call `Entry` or `Reset`. The last entry, however, is a bit more difficult to access. Here is how you can do it:

```
soup := GetUnionSoup("Names");
curs := Query(soup, {type: 'index, indexPath: 'sor-
tOn});

curs:GotoKey("zzzz"); // pick a big key!

// go off the end
while curs:Next() do begin
end;
// back one
e := curs:Prev();
Print(e);
{  cardType: 0,
   company: "Zonnik Corp.",
   sortOn: "Zonnik Corp.",
   address: "25 Houston Drive",
   city: "Filmord",
   region: "OH",
   postal_code: "45150",
   email: "Zonnik@applelink.apple.com",
   phones: ["(513) 555-1212"],
   labels: Business,
   _uniqueID: 108,
   _modtime: 47135291}
```

Finding All Names Matching a String

There are two types of string-matching cursors. The first is `text`. It looks through all slots in an entry. If it finds the specified text anywhere, that entry is part of the cursor. A `words` search differs in two ways:

- The text must be at the beginning of a word. Any other place within the string is ignored.
- You can have multiple words in the query, but each word must be found in some slot in the entry.

Text Search

```
soup := GetUnionSoup("Names");
curs := Query(soup, {type: 'text, text: "liJo"});

e := curs:Entry();
while e do begin
   print(e);
   e := curs:Next();
end;
{  cardType: 0,
   company: "Waterside Productions",
   address: "2191 San Elijo Avenue",
   city: "Cardiff",
   region: "CA",
   postal_code: "92007",
   phones: ["(619) 632-9190", "(619) 632-9295"],
   sortOn: "Waterside Productions",
   labels: Business,
   _uniqueID: 120,
   _modtime: 47221111}
```

Words Search

```
soup := GetUnionSoup("Names");
curs := Query(soup, {type: 'text, words: ["lijo"]});

e := curs:Entry();
while e do begin
   print(e);
   e := curs:Next();
end
doesn't find anything

soup := GetUnionSoup("Names");
curs := Query(soup,{type: 'text, words: ["Elijo"]});

e := curs:Entry();
while e do begin
```

```
      print(e);
      e := curs:Next();
   end;
   {  cardType: 0,
      company: "Waterside Productions",
      address: "2191 San Elijo Avenue",
      city: "Cardiff",
      region: "CA",
      postal_code: "92007",
      phones: ["(619) 632-9190", "(619) 632-9295"],
      sortOn: "Waterside Productions",
      labels: Business,
      _uniqueID: 120,
      _modtime: 47221111}
```

Removing a Store while a Cursor Is Iterating

Question: What happens to a cursor when one of the stores is removed? *Answer:* The cursor just skips over any of the entries that had been on that store. If the current entry happens to be on the removed store, then `cursor:Entry()` returns the symbol `'deleted`.

Here is how you can observe the behavior. Insert a card that contains some names on it. Then, execute the following code from the inspector that iterates through entries until it finds one not on the internal store:

```
soup := GetUnionSoup("Names");
curs := Query(soup, {type: 'index);

e := curs:Entry();
internalStore := GetStores()[0];
while e and EntryStore(e) != internalStore do begin
   e := curs:Next();
end
Print(curs:Entry());
{
   sortOn: ...
   ...
}
```

Now, pull the card from the Newton and execute the `Print` statement again:

```
Print(curs:Entry());
deleted
```

The code you write in your application should always consider the fact that `Entry` may return `deleted`.

Forgetting to Call EntryChange

Modifying entries can be deceptive. If you modify an entry and move forward and back again, you will see the modified entry. This is true even if you do not call `EntryChange` after the modification. The entry will eventually revert to the original, however, making this a deceptive change. Here is what causes the idiosyncrasy: entries are cached in memory, even if there are no references to them. When an entry is no longer in the cache it must be reread from the soup.

To test this, modify an address in the Names soup without calling `EntryChange`:

```
soup := GetUnionSoup("Names");
curs := Query(soup, {type: 'index});

e := curs:Entry();
e.address := "FooBar";
```

Open the Names application and scroll to the item you changed. You should see that the address is changed to "FooBar". Now, close the Names application and reset your Newton. Reopen the Names application and scroll to the item you changed. It should no longer appear as "FooBar", but as its original value.

The moral of this example should be clear: *always call `EntryChange` to update an entry in the soup.*

Indexing on Multiple Slots in an Entry

Imagine you have a soup of student entries containing name and grade slots. First, you wish to sort the entries by grade, and then by name. Here is an example of how you do that using indexes. Three students will suffice for the example:

```
{name: "Gabriel", grade: "A",}
{name: "Alex",    grade: "B",}
{name: "Fran",    grade: "A",}
```

We want the students to be ordered first by grade (A first, then B), and second by name. Given the above students, the order would be Fran, Gabriel, and then Alex.

As there is no way to create an index based on more than one slot, you need to use a workaround. You create a new slot whose value is based on the combined values of the slots you want in the index. This trick gives you the proper result. Given the above student frames, you could add a new slot, `sortSlot`, which contains both the name and the grade:

```
{name:"Gabriel", grade:"A", sortSlot: "AGabriel"},
{name:"Alex",    grade:"B", sortSlot:"BAlex"}
{name:"Fran",    grade:"A", sortSlot:"AFran"}
```

Now you can create an index on the `sortSlot` slot. As you can see, you get the desired index: first by grade, then by name.

Note: Do not forget to update this type of composite slot. If you use a slot to capture information from other slots, it needs updating when the values of its information slots change. In our example, if the `name` or `grade` slot changes, the `sortSlot` needs to change too.

Handling Soups in Your Application

There are several important issues involving the use of soups in an application. From naming your soup to determining when you should remove it, these are all things that you need to know before shipping your final application.

What's in a Soup Name

The soup name must be unique for each application. So, how do you ensure that your name does not collide with the soup name of some other application? Apple Computer has created a system to avoid such collisions. You register a unique name (commonly your company name) with Apple. Apple ensures that the name you register isn't already in use. Naming your soup is then a simple matter. Pick a soup name and append that with a colon and your registered unique name. For further details, see "Creating Unique Application Symbols and Names" on page 344.

Creating Your Soup

An application should create its soup the first time it needs it. Normally, this will be when the application is initially run. To do this, you will create the soup in your base view's `viewSetupFormScript`.

Note: One tricky variation to account for is an entry beamed to your application before the application has ever been run. You would take care of this in your base view's `putAwayScript`.

Don't Create Soups in Your Install Script

Many of you may have considered creating a soup when your application is first installed (in your `InstallScript`). You are better off waiting until the application is run. Just imagine a PCMCIA card with 20 applications on it that you lend to your mother so that she can run one of them on her Newton. If each of the applications creates its soup when it is installed, your mother's Newton now has 19 new soups she will never need. Waiting until your application is run, means mom's Newton only has one new soup. Your user would prefer the latter implementation, as would your mother.

When to Remove Soups

If you create a soup when your application is first run, when should you remove it? Certainly not when your application quits—this is way too soon.

You might be tempted to remove it in `RemoveScript`—the function that handles deinstalling your application. Unfortunately, `RemoveScript` is called both when a user explicitly removes software, and when a card is removed. Currently, your application cannot distinguish between the two different user actions that result in a call to `RemoveScript`. If there were a way, you could ask the user about soup removal at application removal time.

Note: Apple may very well provide a way to tell the difference between the two types of package removal. If that occurs, they will also give direction to developers on the proper soup removal interface to provide to users.

Unfortunately, the proper solution for the time being is somewhat unpalatable. You never remove your soup. Users will need to use a utility soup application to do soup management. The bright side of this is that it does provide a great third-party opportunity.

RegisterCardSoup

```
RegisterCardSoup(soupName, soupIndexes, appSymbol, appObject)
```

There is more to creating a soup than just calling `CreateSoup`. Likewise, there is more to opening a soup than calling `GetUnionSoup` when your application starts. The `RegisterCardSoup` routine does all the additional things.

`RegisterCardSoup` takes four parameters:

soupName	A string that represents the name of your soup. This should be unique (see "Creating Unique Application Symbols and Names" on page 344).
soupIndexes	An array of frames. Each describes a soup index that needs to be created.
appSymbol	The unique application symbol.
appObject	An array of two strings. The first is the singular version of the items in the soup. The second is the plural version (for example: `["Name", "Names"]` or `["Goose", "Geese"]`).

You will normally define constants for each of these four parameters in your "Project Data" file. Here are some examples:

```
constant kAppSymbol := '|WaiterHelper:Calliope|;
constant kPackageName := "WaiterHelper:Calliope";
constant kAppObject := '["Order","Orders"];

constant kSoupName := kPackageName;
constant kSoupIndexes := '[{structure: slot, path:
orderNumber, type: integer}];
```

`ReigsterCardSoup` is not a global function, but is defined as a constant function. Because of this, the way you call it is somewhat unusual. You'll call `RegisterCardSoup` when your application opens. In your base view's `viewSetupFormScript`, you will add a call:

```
call kRegisterCardSoupFunc(kSoupName,kSoupIndexes,
   kAppSymbol, kAppObject);
```

`RegisterCardSoup` returns a union soup, so you do not need to call `GetUnionSoup`.

What RegisterCardSoup Does

When a card is inserted, the Newton system loops through each soup which has been registered by `RegisterCardSoup`. If a particular soup isn't present on the card, the soup is created along with the indexes which were registered for it. A running application can then be sure that the union soup includes a soup on a card. This card soup is vital if new items are to be stored on the card, or if an item is moved to the card (with the action button).

`RegisterCardSoup` ensures that when a store is added, your soup and indexes are added to it. `RegisterCardSoup` also examines all current stores, creating the soup on any (writable) stores where it does not currently exist.

Note: The Newton user who likes to flick from application to application, opening each just to see what is there, is inadvertently adding soups to all existing stores. The Newton is not a television; users have to pay a price for their application surfing. There is unfortunately nothing you, the application designer, can do about this, except perhaps give a warning in your documentation.

UnregisterCardSoup

`UnregisterCardSoup(`*`soupName`*`)`

`UnregisterCardSoup` is the opposite of `RegisterCardSoup`. You call it in your base view's `viewQuitScript` and it is used when your application closes. You, likewise, need to add a slot named `UnregisterCardSoup` to your base view and initialize it to the value `kUnregisterCardSoup`.

In the `viewQuitScript`, you call this routine with the name of your soup:

```
call kUnregisterCardSoupFunc with (soupName);
```

It unregisters the soup name (along with the indexes you've specified). Now, when a card is inserted in the Newton, your soup will not be created on it.

Handling Changes to Your Soup

`soupChanged(soupName)`

The `soupChanged` method is a slot you create in your application's base view template. It takes one parameter, the name of the soup that has changed. You'll use this parameter only if your application adds more than one soup to the `soupNotify` array (see "Soup Change Notification" on page 216 for a description of this array).

`BroadcastSoupChange` sends the `soupChanged` message to all registered applications with soups that get changed. When your application receives the `soupChanged` message, it will normally just redraw (rereading information from the soup). If your application isn't open, it will normally do nothing.

Remember that an application gets registered by adding two entries to the `soupNotify` array: the soup name and the application symbol. `BroadcastSoupChange` messages are sent when the user inserts or removes a card, or another application changes your soup.

Adding Soups to WaiterHelper

There is quite a bit of code to add in this chapter to our sample application. Note, we will frequently suggest that you build and download your application to the Newton to test it. Ignore this advice at your own peril.

Creating the Soup

To begin with, let's create our application soup when our application opens and close it when the application quits. Here is what we need to add:

A `viewSetupFormScript` to the base view:

```
func()
begin
   self.theSoup := call kRegisterCardSoupFunc with
      (kSoupName,kSoupIndexes, kAppSymbol,
      kAppObject);
end
```

A `viewQuitScript` in the base view:

```
func()
begin
   call kUnRegisterCardSoupFunc with (kSoupName);

   // un-needed slots
   self.theSoup := nil;
end
```

So the soup can be garbage collected

Add a `theSoup` slot to the base view. Add definitions for the various constants to the Project Data file:

```
constant kAppSymbol := '|ChangeMe:SIG|;
constant kApplicationName := "WaiterHelper";
constant kHiliteNow := true;
constant kAppObject := '["Bill", "Bills"];
constant kSoupIndexes := '[
   {structure: slot, path: date,type: int},
   {structure: slot, path: tableNumber,  type: int},
   {structure: slot, path: checkNumber, type: int},
];
constant kSoupName := "WaiterHelper:Calliope";
```

Verifying the Soup's Existence

Once we've made these changes, launching our application creates a "WaiterHelper:Calliope" soup. To verify, execute the following in the Inspector:

```
GetUnionSoup("WaiterHelper:Calliope");
```

The union soup should not be `NIL`.

Creating Random Soup Entries

Next, we'll create some random soup entries. Modify the `viewSetupFormScript` of the base view to do a query. Add thirty random bills if the soup is empty:

```
func()
begin
   self.theSoup := call kRegisterCardSoupFunc with
      (kSoupName,kSoupIndexes, kAppSymbol,
      kAppObject);

   // display sorted same way when they last ran, or
```

```
      // sorted by date otherwise
      local path := currentIndexPath;
      if not path then
         path := 'date;
      :CreateCursor(path);

      if not theCursor:Entry() then begin
         // Add some random entries so the user doesn't
         // start out with a blank screen
         for i := 1 to 30 do
            theSoup:AddToDefaultStore(
               :CreateRandomBill((i mod 4) + 1));
         theCursor:Reset();
      end;
   end
```

Modify the `viewQuitScript` of the base view to `NIL` out `theCursor`:

```
   func()
   begin
      call kUnRegisterCardSoupFunc with (kSoupName);

      // un-needed slots
      self.theSoup := nil;
      self.theCursor := nil;
   end
```

Add a `theCursor` slot and a `currentIndexPath` slot to the base view. Add `CreateCursor` to the base view:

```
   func(theIndexPath)
   begin
      local oldEntry;

      if theCursor then
         oldEntry := theCursor:Entry();

      self.theCursor :=
         Query(theSoup, {type: 'index,
            indexPath: theIndexPath});
      self.currentIndexPath := theIndexPath;

      // restore the current entry
      if oldEntry then
         theCursor:Goto(oldEntry);
   end
```

At this point, running your application should create 30 random bills in your soup. Give it a try. The first time you launch it, it will take a few seconds while it creates the entries and stores them in the soup. Subsequent times you open the application should be quicker. Test by writing code in the inspector which creates a query.

Displaying Bills in the Overview

Next, let's display the bills from the soup in the overview.

Making the detail View Invisible

First, make the detail view invisible (edit the `viewFlags` slot). Keep the overview invisible—we'll have the application open it on startup.

Displaying from the Soup

Add a `DisplayBillsFromSoup` method to the `rowContainer` template:

```
func()
begin
   // displays bills from current cursor position
   // without changing the current cursor position
   local clonedCursor := theCursor:Clone();

   foreach child in childrenViews do begin
      child:DisplayBillInRow(clonedCursor:Entry());
      clonedCursor:Next();
   end;
end;
```

Add a `childrenViews` slot to the `rowContainer` template and initialize that slot in the `viewSetupDoneScript` of the `rowContainer` template:

```
func()
begin
   // later we'll be hiding and showing children
   // which changes the order of children returned
   // by ChildViewFrames.  Getting it now and saving
   // it assures us the order won't change
   self.childrenViews := :ChildViewFrames();
end
```

Naming Templates

Name the base template "base", and name the linked subviews "overview" and "detail". Declare them to base.

Declare the rowContainer to the overview template.

In the row layout, declare tableRow, dateRow, checkNumberRow, and billRow to the row template.

Displaying the Overview

Whenever the overview is displayed (either opened or shown), it should display bills from the soup. Add a `viewShowScript` to the overview template to do this:

```
func()
begin
   // whenever the overview is shown, it
   // automatically displays entries from the soup
   rowContainer:DisplayBillsFromSoup();
end
```

When the application opens, it should open to the overview. Add a `viewSetupDoneScript` to the base template:

```
func()
begin
   overview:Open();
end
```

Displaying a Bill in a Row

Declare the `tableRow`, `dateRow`, `checkNumberRow`, and `billRow` templates to their parent, the `row` template. Add a `billInThisRow` slot to the `row` template. Add a `DisplayBillInRow` method to `row`:

```
func(bill)
begin
   self.billInThisRow := bill;
   if bill <> nil then begin
      SetValue(tableRow,       'text,
         NumberStr(bill.tableNumber));
      SetValue(dateRow,        'text,
         DateNTime(bill.date));
      SetValue(checkNumberRow, 'text,
         NumberStr(bill.checkNumber));
```

```
            SetValue(billRow,        'text,
               FormattedNumberStr(
               billFunctions:GetTotal(bill), "%0.2f"));
            if not Visible(self) then
               self:Show();
         end else begin
            // don't display this row if it is empty
            if Visible(self) then
               self:Hide();
         end;
      end
```

This code displays the table number, date, check number, and total. If the bill is `NIL`, however, it hides the row.

At this point, if you run your application, it should display two bills (or more, if you have additional rows in the rowContainer) in the overview; we'll add more row views shortly. Give it a whirl.

Accommodating Various Screen Sizes

Resizing the Base View

We want our application to resize vertically if the screen size is different. To do so, we need to add the following code to the beginning of the `viewSetupFormScript` of the base view:

```
// resize vertically for screen size

local params := GetAppParams();
self.viewBounds := RelBounds(
   viewBounds.left,
   params.appAreaTop + 2, // display frame at top
   viewBounds.right - viewBounds.left,
   params.appAreaHeight - 1
);
```

Creating Rows Dynamically

Now let's determine the number of rows in the rowContainer based on the dynamic size of the rowContainer. Delete the child templates of rowContainer and add a `viewSetupChildrenScript` to the rowContainer template:

```
func()
begin
   local rowHeight := pt_row.viewBounds.bottom;
   local viewHeight := :LocalBox().bottom;
   local numberRows := (viewHeight) div rowHeight;

   //stepChildren is an array of row templates
   self.stepChildren := array(numberRows, pt_row);
end
```

Now when you run your application, it should show enough rows to fill the overview.

Sorting by Table Number and Check Number

Next, let's allow sorting by values other than date. Remember that we created indexes for the table number and check number. Tapping on the header should change the sort order. For example, tapping on the word "Table" should sort by table number.

Making the Heads Clickable

Here are the steps you need to take in the overview template:

1. Add a `viewFlags` slot to `tableStatic`, `dateAndTimeStatic`, and `checkNumberStatic`.
2. Set the `vClickable` flag of each.
3. Add an `indexPath` slot to `tableStatic` with the value `'tableNumber`.
4. Add an `indexPath` slot to `dateAndTimeStatic` with the value `'date`.
5. Add an `indexPath` slot to `checkNumberStatic` with the value `'checkNumber`.
6. Rename tableStatic to tableNumber, dateAndTimeStatic to date, and checkNumberStatic to checkNumber (this way, the template names match the slot names they display from the entry).
7. Declare tableNumber, date, and checkNumber to the header template.

8. Add a `viewClickScript` to each of the three templates:

```
func(unit)
begin
   if :TrackHilite(unit) then begin
      local oldView :=
         GetVariable(self, currentIndexPath);
      :Hilite(nil);
      if oldView <> self then begin
         // change the cursor
         local base := GetRoot().(kAppSymbol);
         base:CreateCursor(indexPath);

         // turn on underlining for this view
         SetValue(self, 'viewFont, underlineFont);

         // turn off underlining for the old view
         SetValue(oldView, 'viewFont,
            ROM_fontSystem9bold);

         // redraw
         rowContainer:DisplayBillsFromSoup();
      end;
   end;

   return true; // we handled the click
end
```

Adding Underline

Add an `underlineFont` slot to the header template with the following value:

```
simpleFont9+tsBold+tsUnderline
```

Add a `viewSetupDoneScript` to the header template to set the underline initially:

```
func()
begin
   SetValue(self.(currentIndexPath), 'viewFont,
      underlineFont);
end
```

At this point, if you run your application, you should be able to tap on the column headers (except Bill) to change the sort order.

Scrolling

To support scrolling, first turn on the `vApplication` bit in the `viewFlags` slot of the overview. This will cause the overview to receive `viewScrollUpScript`, `viewScrollDownScript`, and `viewOverviewScript` messages when the user taps the appropriate button.

Scrolling Down and Up

Add a `viewScrollDownScript` to the overview template:

```
func()
begin
   if theCursor:Clone():Move(1) then begin
      theCursor:Move(1);
      rowContainer:SlideEffect(
         -pt_row.viewBounds.bottom,
         0,
         scrollDownSound,
         'DisplayBillsFromSoup,
         []
      );
   end;
end
```

Add a `viewScrollUpScript` to the same template:

```
func()
begin
   if theCursor:Clone():Move(-1) then begin
      theCursor:Move(-1);
      rowContainer:SlideEffect(
         pt_row.viewBounds.bottom,
         0,
         scrollUpSound,
         'DisplayBillsFromSoup,
         []
      );
   end;
end
```

Finally, set up the rowContainer so that it is an even multiple of the row height by adding a `viewSetupFormScript` to the rowContainer:

```
func()
begin
   // round the height of this view down to
   // an even multiple of the height of each row
   // That way, scrolling down can scroll by the
   // height of a row
   local protoHeight := pt_row.viewBounds.bottom;
   local parentHeight :=
      :Parent():LocalBox().bottom;
   local unmodifiedHeight := parentHeight -
      viewBounds.top + viewBounds.bottom;
   local excess := unmodifiedHeight mod
      protoHeight;
   self.viewBounds := Clone(viewBounds);
   viewBounds.bottom := viewBounds.bottom - excess;
end
```

Now, you should be able to scroll up and down in your application. Give it a scroll.

Tapping on a Row

It is now time to handle tapping on a particular row to switch to that detail view. Set the row proto to `vClickable`, and add a `viewClickScript`:

```
func(unit)
begin
   if :TrackHilite(unit) then begin
      :Hilite(nil);
      local base := GetRoot().(kAppSymbol);
      base:DisplayDetail(billInThisRow);
   end;

   return true; // we handled the click
end
```

Add a `DisplayDetail` method to the base template:

```
func(bill)
begin
   // if bill is nil, display the current entry

   if not bill then begin
      bill := theCursor:Entry();
      if not bill then
         theCursor:Reset();
      bill := theCursor:Entry();
```

```
        end;
        if bill then begin
           // make sure cursor points to displayed bill
           if theCursor:Entry() <> bill then
              theCursor:Goto(bill);

           if Visible(overview) then
              overview:Hide();
           if not GetView(detail) then
              detail:Open() // should only happen once
           else if not Visible(detail) then
                 detail:Show()
           else
              detail:DisplayBill(bill);
        end;
     end
```

We no longer want to display a random entry when the detail view is opened. Remove the `viewSetupDoneScript` of the detail template.

Finally, when the detail view is shown, it should display the current cursor entry. To do so, add a `viewShowScript` slot to the detail template:

```
func()
begin
   :DisplayBill(theCursor:Entry());
end
```

Now run your application, and you should be able to tap on a row to display the detail view for that bill.

Supporting Tapping to Switch between Views

The overview button is another way to switch between the overview and the detail view. Add a `viewOverviewScript` to overview:

```
func()
begin
   local base := GetRoot().(kAppSymbol);
   base:DisplayDetail(nil);
end
```

To support the overview button from the detail template, set the `viewFlags` to `vApplication`, and add a `viewOverviewScript`:

```
func()
begin
   local base := GetRoot().(kAppSymbol);
   base:DisplayOverview();
end
```

Add a `DisplayOverview` method to the base template:

```
func()
begin
   if Visible(detail) then
      detail:Hide();
   if not GetView(overview) then
      overview:Open();   // should only happen once

   if not Visible(overview) then
      overview:Show();
end
```

At this point, you can run your application; tapping the overview button toggles you between the overview and the detail view. Still to come: scrolling in the detail view, saving a changed bill, creating a new bill, and toggling back to the correct view and location. Let's tackle the last part first.

Reopening to the Same Display

The user would like the application to reopen to the same display, looking at the same data. The application needs to save what entry the cursor is on, and whether the template or overview is open. To do this, you add two slots to the base template: `openToDetail` and `entryToDisplay`.

Save this information in the `viewQuitScript` of the base template:

```
func()
begin
   call kUnRegisterCardSoupFunc with (kSoupName);

   local theEntry := theCursor:Entry();
   // re-open to same display
   self.openToDetail := Visible(detail);
   // save a triplet to identify the entry
   self.entryToDisplay := {
      date:     theEntry.date,
      entryID:  EntryUniqueID(theEntry),
      soupID:   EntrySoup(theEntry):GetSignature(),
      storeID:  EntryStore(theEntry):GetSignature(),
   };
```

```
      // un-needed slots
      self.theSoup := nil;
      self.theCursor := nil;
   end
```

Next, read that information in the `viewSetupDoneScript` of the base template:

```
   func()
   begin
      if entryToDisplay <> nil then begin
         local e;
         local q;

         // create query of all entries on
         // specified date
         q := Query(theSoup,
            {type: 'index,
            indexPath: 'date,
            startKey: entryToDisplay.date,
            endTest:
               func(e)
                  e.date <> entryToDisplay.date
            });

         // for each entry on the date,
         // see if has correct triplet of
         // unique id's.
         e := q:Entry();
         while e do begin
            if (EntryUniqueID(e) =
               entryToDisplay.entryID) and
               (EntrySoup(e):GetSignature() =
               entryToDisplay.soupID) and
               (EntryStore(e):GetSignature() =
               entryToDisplay.storeID) then begin
               theCursor:Goto(e);
               break;
            end;
            e := q:Next();
         end;
      end;

      if openToDetail then
         detail:Open();
      else
         overview:Open();
   end
```

Now if you run your application, it should open to the same view and bill displayed when it was last closed.

Scrolling in the Detail View

Add a `viewScrollUpScript` to the detail template:

```
func()
begin
   local e := theCursor:Prev();
   if e then begin
      local height := :LocalBox().bottom;
      :SlideEffect(
         // 1 and 1 each for top & bottom inset
         height + 3,
         0,
         scrollUpSound,
         'DisplayBill,
         [e]
      );
   end else
     theCursor:Next();
end
```

Then, add a `viewScrollDownScript`:

```
func()
begin
   local e := theCursor:Next();
   if e then begin
      local height := :LocalBox().bottom;
      :SlideEffect(
         // 1 and 1 each for top & bottom inset
         -(height + 3),
         0,
         scrollDownSound,
         'DisplayBill,
         [e]
      );
   end else
     theCursor:Prev();
end
```

Now run your application; notice you can scroll in the detail view.

Saving Changes to a Bill

Now it is time to save the changes to a bill. There are three steps to this:

1. Make sure that each of the controls or inputs in the detail view modifies `currentBill`.
2. Set a flag marking that `currentBill` has been changed.
3. Save the actual change whenever the user leaves the detail view and that bill.

There are three ways for the user to leave the bill: when a different bill is displayed in the detail view (for example, when the user scrolls), using the overview button to switch to the overview, and quitting the application.

Flagging Whether the Bill Has Changed

Add a `billHasChanged` slot to the detail template, and add a `BillChanged` method to the detail template:

```
func()
begin
   billHasChanged := true;
end
```

We need to initialize `billHasChanged` to `NIL` when a bill is displayed. Modify `DisplayBill` in the detail template to:

```
func(bill)
begin
   // make bill accessible to children
   self.currentBill := bill;

   SetValue(title, 'text, "Check #" &
      NumberStr(bill.checkNumber));
   SetValue(date, 'text, DateNTime(bill.date));
   numPeople:UpdateText(
      NumberStr(bill.numberInParty));
   tablePicker:UpdateText(
      NumberStr(bill.tableNumber));
   table:SelectChairNumber(0, not kHiliteNow);

   billHasChanged := nil;
end
```

Now, let us call `BillChanged` at each point a user action changes the bill. Modify the `textChanged` method in the commentPicker template to:

```
func()
begin
   local newComment;
   local hasChanged := nil;

   if entryLine.text = nil or
      StrLen(entryLine.text) = 0 then
      newComment := nil;
   else
      newComment := entryLine.text;

   // check for one NIL, and one non-NIL
   hasChanged :=
      (not newComment) <> (not currentItem.comment);

   if not hasChanged and newComment and
      currentItem.comment then
      hasChanged := not
         StrEqual(newComment, currentItem.comment);

   if hasChanged then begin
      currentItem.comment := Clone(newComment);
      order:ItemChanged(currentItem);
      detail:BillChanged();
   end;
end
```

Add a `labelActionScript` method to the itemPicker, the categoryPicker, the tablePicker, and the numPeople templates. This method will be called when the user selects an item from the picker (`textChanged` is called anytime the text changes, not just when the user changes it):

```
func(cmd)
begin
   detail:BillChanged();
end
```

Add a call to `BillChanged` in `AddNewItem` in the order template:

```
func()
begin
   if Length(currentOrder) < 5 then begin
      newItem := {itemSymbol: 'none};
      AddArraySlot(currentOrder, newItem);
      newView := AddStepView(self, pt_itemRow);
      newView:Displayitem(newItem);
```

```
        :DisplayOneItem(newView, kHiliteNow);
        detail:BillChanged();
      end else
        :Notify(kNotifyAlert,
           EnsureInternal(kApplicationName),
           EnsureInternal("Can't add more items."));
end
```

Add a call in `DeleteSelectedItem` in the order template:

```
func()
begin
   if Length(currentOrder) > 1 then begin
      SetRemove(currentOrder, selectedItem);
      // re-create views
      :DisplayOrder(currentOrder, kHiliteNow);
      detail:BillChanged();
   end else
      :Notify(kNotifyAlert,
         EnsureInternal(kApplicationName),
         EnsureInternal(
         "Can't delete the last item."));
end
```

Finally, the order can change if the user enters a new date. Add a `viewChangedScript` method to the date template:

```
func(slot, view)
begin
   if slot = 'text then begin
      local num := StringToDate(text);

   if num then begin
      currentBill.date := num;
      detail:BillChanged();
   end;
   end;
end
```

Saving Changes

Let's add a method, `SaveCurrentBill`, to the detail template:

```
func()
begin
   if currentBill and billHasChanged then begin
      EntryChange(currentBill);
   end;
   currentBill := nil;
   billHasChanged := nil;
end;
```

We will call `SaveCurrentBill` from `DisplayBill` (before displaying a new bill):

```
func(bill)
begin
   :SaveCurrentBill();

   // make bill accessible to children
   self.currentBill := bill;

   SetValue(title, 'text, "Check #" &
      NumberStr(bill.checkNumber));
   SetValue(date, 'text, DateNTime(bill.date));
   numPeople:UpdateText(
      NumberStr(bill.numberInParty));
   tablePicker:UpdateText(
      NumberStr(bill.tableNumber));
   table:SelectChairNumber(0, not kHiliteNow);

   billHasChanged := nil;
end
```

We'll also add a `viewHideScript` to the detail template. This will be called when we switch back to the overview:

```
func()
begin
   :SaveCurrentBill();
end
```

Finally, we'll add a `viewQuitScript` to the detail template, which will be called when the application is closed:

```
func()
begin
   :SaveCurrentBill();
end
```

Run your application and notice that changes to a bill are saved—even if you reset!

Creating a New Bill

We'll create new bills with a "New" button at the bottom of the application, in the status bar (deleting is done from the action button, which isn't covered in this book). We need to add the button as a child of the status bar. Unfortunately, we can't do this using protoApp, since it provides no access to its status bar.

Changing the protoApp into a clView

Instead, we'll trade a protoApp for a clView. Here are the steps:

1. First, create `viewJustify` and `viewFlags` slots in the base template (this will create slots with the default protoApp values).
2. Change the `_proto` slot of the base template to clView.
3. Draw a `protoTitle` at the top of the base template. Set its `title` to "WaiterHelper", and its `viewBounds` to `{left: 0, right: 80, top: 0, bottom: 16}`.
4. Delete the `title` slot from the base template.
5. Draw a protoStatus (not a protoStatusBar, which has no close box) in the base template. It should automatically draw itself at the bottom of the base template.
6. The final modification to the base template is to add a `declareSelf` slot. Set it to `'base`. This is similar to declaring; at run time it adds a `base` slot to the base which points to itself. This is used by the close box, whose action when tapped is to call `base:Close()`.

At this point, you can build and download. The change from a protoApp to a clView shouldn't be noticeable.

Adding a New Button

Now we need to add the New button.

1. Draw a `protoTextButton` within the `protoStatus`. Name it "newButton".

2. Set its viewBounds to `{left: 25, right: 55, top: 2, bottom: 15}`.

3. Set its text slot to "New".

   Modify its buttonClickScript to:

```
func()
begin
   local base := GetRoot().(kAppSymbol);
   base:AddNewBill();
end
```

   Add an AddNewBill method to the base template:

```
func()
begin
   local newBill;

   newBill := {
      numberInParty: 0, // set by SetNumberInParty
      date: Time(),     // now
      tableNumber: 51,
      checkNumber: :HighestCheckNumber() + 1,
      orders: [],
   };
   billFunctions:SetNumberInParty(newBill, 1);

   theSoup:AddToDefaultStore(newBill);
   :DisplayDetail(newBill);
end
```

To calculate the highest check number, use the brute force approach: add a HighestCheckNumber method to the base template:

```
func()
begin
   local cursor;

   cursor := Query(theSoup,
      {type: 'index,
       indexPath: 'checkNumber
      });
   cursor:GotoKey(999999); // a big number
```

```
        // go off the end
        while cursor:Next() do
        begin
           // do nothing
        end;
        // back up one
        local e := cursor:Prev();
        if e then
           return e.checkNumber;
        else
           return 0;
     end
```

It utilizes the index on `checkNumber` to find quickly the highest check number.

Run your application and you should be able to create new bills. Each bill's check number should be one more than the then-highest check number.

Handling Soup Changes

We need to respond to soup changes (for instance, when a card is inserted or removed). To do so, we must first add entries to the `soupNotify` array. Modify the `viewSetupFormScript` of the base template:

```
func()
begin
   // resize vertically for screen size

   local params := GetAppParams();
   self.viewBounds := RelBounds(
      viewBounds.left,
      params.appAreaTop + 2, // display frame at top
      viewBounds.right - viewBounds.left,
      params.appAreaHeight - 1
   );

   self.theSoup := call kRegisterCardSoupFunc with
      (kSoupName,kSoupIndexes, kAppSymbol,
      kAppObject);

   // display sorted same way when they last ran, or
   // sorted by date otherwise
   local path := currentIndexPath;
   if not path then
      path := 'date;
   :CreateCursor(path);
```

```
    if not theCursor:Entry() then begin
       // Add some random entries so the user doesn't
       // start out with a blank screen
       for i := 1 to 30 do
          theSoup:AddToDefaultStore(
             :CreateRandomBill((i mod 4) + 1));
       theCursor:Reset();
    end;

    // be notified of soup changes
    AddArraySlot(soupNotify, kSoupName);
    AddArraySlot(soupNotify, kAppSymbol);
end
```

Modify the `viewQuitScript` of the base template:

```
func()
begin
   call kUnRegisterCardSoupFunc with (kSoupName);

   local theEntry := theCursor:Entry();
   // re-open to same display
   self.openToDetail := Visible(detail);
   // save a triplet to identify the entry
   self.entryToDisplay := {
      date:    theEntry.date,
      entryID: EntryUniqueID(theEntry),
      soupID:  EntrySoup(theEntry):GetSignature(),
      storeID: EntryStore(theEntry):GetSignature(),
   };

   // un-needed slots
   self.theSoup := nil;
   self.theCursor := nil;

   // no more notification of soup changes:
   local soupNotifyPos
      := ArrayPos(soupNotify, kAppSymbol, 0, nil);
   ArrayRemoveCount(soupNotify,soupNotifyPos -1,2);
end
```

Now we must add a `soupChanged` method to the base template:

```
func(theSoupName)
begin
   // change the cursor so it doesn't
   // point at 'deleted
   local e := theCursor:Entry();
   if e = 'deleted then begin
      e := theCursor:Next();
         if e = nil then
            e := theCursor:Prev();
   end else if e <> nil then
      EntryUndoChanges(e); // re-read from soup
   else begin
      theCursor:Reset();    // could have been
      e := theCursor:Entry()// completely empty soup
   end;

   if Visible(overview) then
      overview:DisplayFromSoup();
   else begin
      if e = nil then
         :DisplayOverview();
      else
         :DisplayDetail(e);
   end;
end
```

This method must watch out for a deleted cursor entry and deal with the case where some entries are created where none existed. There is also an edge case that we have to deal with: when all the entries are deleted we must switch back to the overview.

Add a `DisplayFromSoup` method to the overview template:

```
func()
begin
   rowContainer:DisplayBillsFromSoup();
end
```

Now you should be able to install your application on the internal store and create some bills on a memory card. When you remove the card, the application should update the screen correctly.

Summary

In this chapter we presented Newton's new form of data storage. You learned how this new model simplifies data for the user. Data is transparent across all applications: enter it once, available to everyone. After discussing the model in general, we covered the various methods that you need to implement data reading and writing. Last of all we implemented soups in our sample application, WaiterHelper.

Chapter 9

Debugging Your Application

Calamity is virtue's opportunity.

—*Seneca*

Without the proper tools and techniques for debugging, application development would be a nightmare. While the debugging tools for the Newton are modest in number, they are nevertheless quite useful. You can do everything from snooping around inside a running application (including accessing all of its views and data) to looking in functions with errors. Further, because the Newton is interactive, you have immediate access to your code.

This chapter will discuss most of these important debugging tools and techniques and help clarify how you will use these tools in your own application development. Let us begin with the single most essential tool, the Inspector.

The Inspector

You will find that the Inspector is the most powerful and flexible tool in your debugging arsenal. It is useful for a wide range of tasks: everything from a place to try out NewtonScript code before implementing it in an application, to a window that gives you access to both data and the views of an application.

To use the inspector, you must:

- Tether your Macintosh to your Newton.
- Make sure that NTK is properly installed.
- Establish a live connection between the Newton and Macintosh (see "The Inspector" on page 380).

Evaluating NewtonScript Code

The Inspector sends NewtonScript code to the Newton, and after it's evaluated you receive the result. To evaluate code, you use the Enter key. When you press the "Enter" key, the selected code block (or the line where the insertion point rests) is compiled on the Macintosh and then downloaded to the Newton. The code is executed there, and then the return result is displayed in the Inspector window.

⚠ *Caution:* As we said, in order to execute NewtonScript code from the Inspector, you use the "Enter" key, not the "Return" key. Pressing the "Return" key in the Inspector just gives you a new line. Veteran Inspector users joke that the "Return" key is a shortcut for the three-character sequence "Return", "Delete", "Enter".

Example

Now, we will look at an example of evaluating code in the Inspector window. In order to calculate the sum of 1, 2, 3, 4, and 5, type:

```
1 + 2 + 3 + 4 + 5
```

into the Inspector. You can evaluate the code in one of two ways:

- Select the entire line and press the "Enter" key (see Figure 9.1).
- Press the "Enter" key without selecting the line (see Figure 9.2).

In both cases, the code is compiled and downloaded to the Newton, where it is interpreted, and then the result, 15, is printed in the Inspector (see Figure 9.3).

.

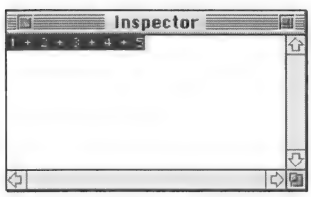


Figure 9.1 Executing the selected NewtonScript.

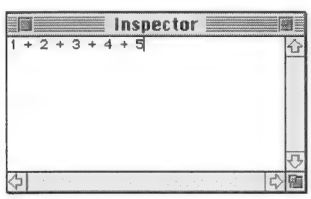


Figure 9.2 Executing the NewtonScript on the line with the insertion point.

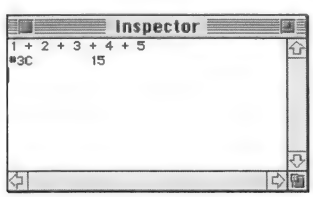


Figure 9.3 Results of executing NewtonScript code.

As you can see in Figure 9.3, the Inspector prints a hexadecimal number before printing the actual value, 15. This hexadecimal number is the 32-bit value representing the number 15 and is not something that should cause you further concern.

Note: You may want to stop the Inspector from printing out the value of the code it is executing (for instance, you may be assigning a large frame or array to a variable). To suppress the execution echo, add a `NIL` statement at the end of your code line or block. Since the Inspector prints out the value of the last statement, only `NIL` will be echoed.
For example, instead of executing

```
x := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
```

execute:

```
x := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]; NIL;
```

Restrictions

Execution Environment

When you are evaluating code from within the Inspector, the execution environment has some fairly unusual characteristics:

- `self` is the globals frame (the result of `GetGlobals()`).
- Any variables you assign to are created as slots in the globals frame.

Thus, when you are in the Inspector and you execute:

```
x := 1 + 2 + 3 + 4 + 5
```

you have actually created a slot named `x` with the value `15` in the globals frame. This slot will remain until you explicitly remove it (with `RemoveSlot`), or until you reset the Newton.

Caution: Don't accidentally assign to variables that already exist as global variables. For instance, don't assign to the variable `functions` since that is an existing (very important) slot in the globals frame.

Functions

If you create a function from within the Inspector, such as:

```
Square := func(x) return x * x;
```

there are two ways to call it. One is to use `call` syntax:

```
call Square with (3);
```

The other is to use message sending syntax. Since `self` is the globals frame, and the `Square` function is now a slot (method) in the globals frame, you can send the `Square` message to `self` with the following:

```
:Square(3);
```

Note: Another alternative is to use a global function declaration:

```
func Square(x) return x * x;
```

and then call the function with:

```
Square(3);
```

Loops

Unfortunately, the looping constructs (`for`, `foreach`, `loop`, `while`, `repeat`) do not work directly within the Inspector. So even though you would like it to, this code won't work:

```
for i := 1 to 5 do
   total := i + total;
```

Where the direct approach fails, the indirect succeeds. You can create functions that contain loops and then call those functions. Thus, to execute the above `for` loop, you can simply nest it within `MyFunction` and then call it:

```
MyFunction := func()
begin
   for i := 1 to 5 do
      total := i + total;
   return total;
end;
:MyFunction();
```

Printing

There are three pretty-printing routines that will print out values to the Inspector window. While each one can print arrays and frames in a readable manner, they differ in small details as shown in Table 9.1.

	`Print`	`Display`	`Write`
Appends a newline after printing.	X		
Surrounds strings with double quotes (e.g., `"Hello"`).	X	X	
Prints dollar sign before characters (e.g., `$H`).	X	X	

Table 9.1 Differences between `Print`, `Display`, and `Write`.

Example

Here are some examples that demonstrate these differences between `Print`, `Display`, and `Write`. The following code for `Print`:

```
Print("Hello"); Print($W); Print("orld");
```

produces:

```
"Hello"
$W
"orld"
```

Using `Display` instead:

```
Display("Hello"); Display($W); Display("orld");
```

produces:

```
"Hello"$W"orld"
```

Finally, using `Write` for the same text:

```
Write("Hello"); Write($W); Write("orld");
```

produces:

```
HelloWorld
```

Controlling the Depth of Printing

The Inspector allows you to print recursively. For instance, if a frame contains a slot that holds a frame or array, you can use the Inspector to print both the enclosed frame and the enclosing frame. This works for the full depth of a structure. As you can imagine, this may or may not be what you intended. Picture displaying `GetRoot()` with no limit on the printing! Fortunately, there is a mechanism for controlling the depth of this recursive printing.

The `printDepth` global variable contains an integer that sets the depth of recursive printing. The higher the number, the more levels that print. The default value for this variable is 1. The following frame serves as a good test for showing these different levels:

```
f := {
   name: "Neil",
   height: 73.25,
   children: [
      {
         name: "Nicholas",
         height: 42,
         children: [],
      },
      {
         name: "Alexander",
         height: 40,
         children: [],
      }
   ]
}
```

Using print depths from -1 to 2, Table 9.2 shows that each change yields one more level of detail. It would take a print depth of 3 in order to see the full contents of the example frame.

Note: If you print out an application's base view without changing the default value for `printDepth`, it will print out all the slots in `GetRoot()` since that is the parent view. To prevent this voluminous listing, set `printDepth` to 0 before printing an application's base view.

PrintDepth -1	*PrintDepth 0*	*PrintDepth 1*	*PrintDepth 2*

`Print(f);`
`{#4413AE9}`
	`Print(f);`
	`{name: "Neil",`
	`height: 73.2500,`
	`children:[#4413AD1]}`
		`Print(f);`
		`{name: "Neil",`
		`height: 73.2500,`
		`children:[{#4413A91}, {#4413AB9}]}`
			`Print(f);`
			`{name: "Neil",`
			`height: 73.2500,`
			`children: [{name: "Nicholas",`
			`height: 42,`
			`children: [#4413A81]},`
			`{name: "Alexander",`
			`height: 40,`
			`children: [#4413AA9]}]}`

Table 9.2 Changing the print depth.

Tracing

If you turn tracing on, execution information about your program will print out in the Inspector. You can set tracing to display either function and variable information, or function information only .

Turning Tracing On and Off

The `trace` global variable controls whether tracing is on or off. To turn tracing on, set `trace` either to the symbol `'functions` or to `true`. The former traces only function calls, while the latter also prints out the variables as they are accessed.

Example

We will use the following code for our trace examples:

```
func()
begin
   local j := self.title;
   :Bar();
   :Foo();
end
```

With `trace` set to `'functions`, the Inspector prints (along with quite a bit of other information):

```
        Sending Bar(5) to #4410D81
        => 1
        Sending Foo() to #4410D81
        => NIL
    => NIL
```

As you can see, the Inspector displays the message name, parameters, the object getting the message, and the return result. The final line shows the return result of the entire function.

As you would expect, with `trace` set to `true`, the inspector prints more:

```
get #4412199 / #60084DC5.title = #600850A5
            Sending Bar(5) to #4412199
            => 1
            Sending Foo() to #4412199
            => NIL
        => NIL
```

The `get` line signifies an access to a slot. The number before `/` is the address of the frame where the search begins (`self` in this case). The number just after `/` is the address of the frame in which the slot was actually found (they might well be different due to inheritance). The value found in the slot follows the `=`. The format, therefore, is:

```
get frameSearched / frameSlotFoundIn.slotName = slotValue
```

Gotchas with Tracing

The real trick to tracing is knowing how to turn it off.

Turning Tracing Off

When tracing is on, copious amounts of information print out in the Inspector. So much prints out, that you probably cannot type in `trace := nil` and execute it as one line before the Inspector spews forth some data in the middle of your attempt. What's an honest programmer to do? Cheat, of course. NTK provides a button in the lower left corner of the inspector which sends `trace := nil` to the Newton.

Tracing Doesn't Take Effect Immediately

When you turn tracing on, trace output does not occur immediately, but rather takes a few seconds. This means that you need to turn tracing on well in advance of the code you are interested in, not immediately before it.

Note: Although it is inconvenient having tracing delayed, it was the right design approach to take. If tracing took place immediately, then the NewtonScript byte code interpreter would have to check the value of trace on every instruction rather than only every so often. The trade-off—speedier code execution versus immediate tracing—is worth it.

Debugging Functions

There are a number of debug functions that return views matching certain criteria or that print out a hierarchy of views. If you call a function that returns a view, the Inspector will also print out the slots in that view. You can also assign the function result to a variable, and thereby have access to the view later.

The debugging functions also provide you with a key to the whole inheritance chain. You can call a function that returns a view in which you can follow the proto chain with the `_proto` slot, the parent chain with the `_parent` slot, and its children with the `ChildViewFrames` method. The entire inheritance structure is just a keystroke away.

Debug

```
Debug(string)
```

This routine takes a string as a parameter, and then searches through the open views for a slot named `text` or a slot named `debug`. The routine searches all such slots until it finds one that contains a value that matches *`string`*. The view that contains the matching value is returned.

Note: For each view, the `Debug` function tries to access a `text` or `debug` slot. Since slot access uses proto inheritance, the `text` or `debug` slot can be in the template, or the proto, and need not be directly in the view frame.

Searching the Text Slot

Since `Debug` searches for a `text` slot, you have access to the large number of views that contain `text` slots. As an example, consider an application that contains a "Press Me" button (see Figure 9.4). You can find that button view from within the Inspector by executing:

```
theButtonView := Debug("Press Me");
```

It is also helpful that you do not need to match the string in question fully. The `Debug` function checks whether its argument is a prefix. Thus, you could execute:

```
theButtonView := Debug("Press");
```

The only problem with less exact matching is that another view with a matching value might be returned instead. For instance, if another view is open that contained a text slot with the value "Press On", `Debug` might return that view in the above case, since "Press" is a prefix of both "Press Me" and "Press On".

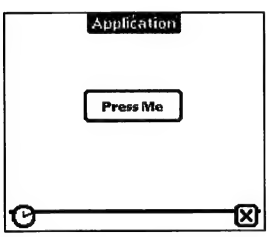


Figure 9.4 An application with a "Press Me" button.

Protos Containing Text Slots

A number of protos contain `text` slots. The protos that use the `text` slot are:

`protoLabelInputLine`	The text in the input field.
`protoInputLine`	The text in the input field.
`protoTextButton`	The name of the button.
`protoLabelPicker`	The label of the picker.
`protoCheckBox`	The name of the checkbox.
`protoRadioButton`	The name of the radio button.
`protoStaticText`	The text shown.
`protoGlance`	The text shown when tapped.

The first two, `protoLabelInputLine` and `protoInputLine`, are of further interest as you can also search for a view containing user-entered text. For instance, consider an application with a `protoInputLine` in which the user has entered the string "Hello world" (see Figure 9.5). You can find that view using:

```
theInputLine := Debug("Hello world");
```

Figure 9.5 A `protoInputLine` containing the string "Hello world".

Searching the Debug Slot

The `Debug` function also searches slots named `debug`. This can be quite useful as you can have NTK automatically create `debug` slots for templates. Here are the conditions under which NTK will do so:

- The project is built for debugging. "Debug Build" on page 378 discusses how to build a project this way.
- The template is named. See "Naming Templates" on page 365.
- The template doesn't already contain a `debug` slot.

Building your project for debugging makes for the easy retrieval of specific views using the template names you specified in NTK. For instance, you can find a cluster view using the name of its template:

```
theCluster := Debug("myRadioCluster");
```

GetView

```
GetView(template)
GetView('viewFrontMost)
GetView('viewFrontMostApp)
GetView('viewFrontKey)
```

This function returns a view based on its parameter:

template	Searches all open views to find one whose template is *template*.
`'viewFrontMost`	Returns the frontmost view with the `vApplication` bit set.
`'viewFrontMostApp`	Returns the frontmost view with the `vApplication` bit set, while skipping views with the `vFloating` bit set.
`'viewFrontKey`	Returns the frontmost view that accepts keystrokes.

The following application clarifies the distinctions between these latter three parameters. It consists of a `protoApp` containing a `protoFloatNGo` with two `protoInputLines` (see Figure 9.6). The `viewFlags` of the `protoFloatNGo` have been altered to `vVisible`, `vApplication`, `vReadOnly`, and `vFloating` (the default is `vReadOnly` and `vFloating`).

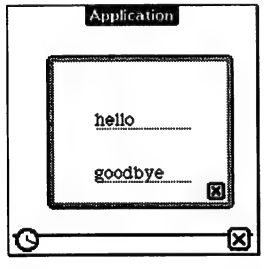


Figure 9.6 An application demonstrating the different uses of `GetView()`.

With the application open:

`GetView('viewFrontMost)` returns the `protoFloatNGo` view.
`GetView('viewFrontMostApp)` returns the `protoApp` view.

After closing the `protoFloatNGo`:

`GetView('viewFrontMost)` returns the `protoApp` view.
`GetView('viewFrontMostApp)` returns the `protoApp` view.

When the application is first opened:

`GetView('viewFrontKey)` returns `NIL`.

Tapping or writing in the first input line activates that view as the target for keystrokes (where keyboard input is directed). At this point:

`GetView('viewFrontKey)` returns the "hello" input view.

Tapping in the other input line switches the target:

`GetView('viewFrontKey)` returns the "goodbye" input view.

DV

`DV(view)`

The `DV` function takes a view as a parameter and prints information about that view, along with all its children, in a compact one-line-per-view format. You also get a quick flash of the view on the Newton—making it more easily identifiable.

Consider as an example a `protoFloatNGo` named "myFloatNGo" containing a `protoStaticText` named "myStaticText". Executing:

```
floater:= Debug("myFloatNGo");
DV(floater);
```

produces the following output:

```
myFloatNGo      #4411C21 [32,128,224,232] 10000041 vVisible vFloating vHasChildrenHint
|protoCloseBox  #4412E89 [210,218,223,231] 40000203 vVisible vReadOnly vClickable
|myStaticText   #4412EA1 [78,198,166,214] 40000003 vVisible vReadOnly
```

The whole `protoFloatNGo` flashes momentarily (see Figure 9.7) and then the output appears in the Inspector. The vertical bar (|) at the left of each line shows you the nesting level. Notice that you also see the `viewbounds` (in screen coordinates), along with a `viewFlags` summary.

Figure 9.7 DV causes the `protoFloatNGo` view to flash.

Using the same example, executing

```
DV(Debug("myStaticText"));
```

produces the following output:

```
myStaticText  #4412EA1 [78,198,166,214] 40000003 vVisible vReadOnly
```

In addition to the output, the static text also gives the comforting flash (see Figure 9.8).

Figure 9.8 DV causes the `protoStaticText` view to flash.

Exceptions

On the Newton system functions throw exceptions, rather than returning error codes. When an exception occurs, the currently executing function aborts, as does the function that calls it, and so on up the line. Eventually, the Newton puts up a slip displaying the error number (see Figure 9.9).

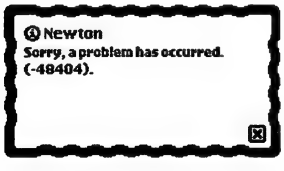


Figure 9.9 Example of Newton displaying error code in response to an exception.

Note: This book does not attempt to cover programming with exceptions in NewtonScript. Instead, it deals with debugging in the face of exceptions.

Breaking on an Exception

By the time the Newton puts up a slip displaying the error number, you have already lost the most valuable information available to you, that is, the context of where the error occurred. When you are debugging, you obviously need as much context as possible. In particular, you need to know what function caused the error—enter the lifesaver, `breakOnThrows`. When the `breakOnThrows` global variable is true, the Newton doesn't abort the currently executing function on an exception, but rather enters a *break loop* (see Figure 9.10).

Note: `breakOnThrows` sounds suspiciously like the title of a Doors song: "Break on Through (to the Other Side)." Who cares? You will when you try to remember the name of the variable.

Once you are in the break loop, the Newton stops handling user input and only responds to commands sent via the Inspector. This, as you will see, comes in quite handy (in much the same way that a thorn-studded rose is still worth holding).

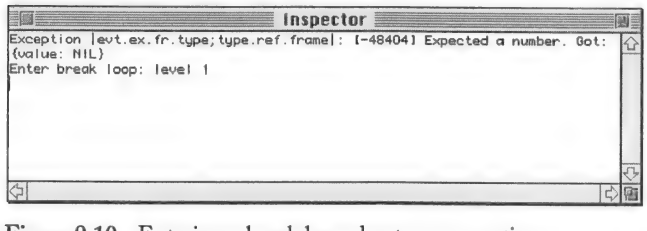


Figure 9.10 Entering a break loop due to an exception.

The Thorns of Break Loops

If `breakOnThrows` is set and an exception occurs while the Newton is in a break loop, it will enter a new break loop. In fact, each and every time you generate an exception you will enter yet another break loop. So that you will know how deep you are, the Inspector prints the level of the break loop (see Figure 9.11).

```
Inspector
Exception |evt.ex.fr.type;type.ref.frame|: [-48404] Expected a number. Got:
{value: NIL}
Enter break loop: level 1

Primmt("Hello")
Exception |evt.ex.fr.intrp;type.ref.frame|: [-48808] Undefined global
function "Primmt"
{Symbol: Primmt}
Enter break loop: level 2
```

Figure 9.11 Entering a second level of a break loop.

The most common cause of further exceptions is mistyping a variable or function name from within the Inspector.

Breaking Using BreakLoop()

Executing the `BreakLoop` global function also puts you in a break loop. Typically, you will embed this call in a function in which you're having problems. Once you are in the break loop, you can look at the values of parameters, local variables, and so forth to debug the problem.

Exiting a Break Loop

```
ExitBreakLoop()
```

To leave a break loop, execute the `ExitBreakLoop()` global function. If you have entered multiple levels of break loops, you'll need to execute `ExitBreakLoop` multiple times.

If you entered the break loop due to an exception, you leave via the final `ExitBreakLoop`. The Newton will then display the standard error slip.

It can be hard to remember whether or not you have exited a break loop when you have pages of listings displayed in the Inspector and you just finished fixing a bug. As a last result, you can follow these steps as a surefire path to exit all current break loops:

1. Keep executing `ExitBreakLoop` until you generate the exception "Not in a break loop".
2. If `breakOnThrows` was set, then you're now in a first-level break loop, and you need to execute `ExitBreakLoop` one more time.

Showing a Stack Trace

Normally, upon entering a break loop you will want to know what function generated the error. The `StackTrace()` global function prints out the stack of functions in progress and contains the particular function in which the exception was thrown.

Example

Here is a sample result of calling `StackTrace()` while in a break loop. The error-generating action was pressing a button:

```
StackTrace()
[{class: StackFrameInfo,
  CodeBlock: NIL,
  programCounter: 2,
  receiver: NIL,
  contextFrame: [local1InD]},
{class: StackFrameInfo,
  CodeBlock: "functions.breakLoop",
  programCounter: NIL,
  receiver: NIL,
  contextFrame: [local1InD]},
{class: StackFrameInfo,
  CodeBlock: MethodD,
  programCounter: 6,
  receiver: NIL,
  contextFrame: [local1InD]},
 {class: StackFrameInfo,
  CodeBlock: MethodC,
  programCounter: 3,
  receiver: NIL,
  contextFrame: [param1InC, param2InC]},
 {class: StackFrameInfo,
  CodeBlock: MethodB,
  programCounter: 7,
  receiver: NIL,
  contextFrame: [param1InB, local1InB, local2InB]},
 {class: StackFrameInfo,
  CodeBlock: buttonClickScript,
  programCounter: 14,
  receiver: NIL,
  contextFrame: [local1, local2]},
 {class: StackFrameInfo,
  CodeBlock: viewClickScript,
  programCounter: 10,
  receiver: NIL,
  contextFrame: [unit]}]
```

Ignore above the line

`StackTrace()` returns an array of frames. Each frame corresponds to a function call currently in progress. The first entry is the most recent function and the last entry is the bottom function on the stack. You can ignore the first two entries in the array since they correspond to functions handling the break loop. *The third entry is the function that caused the Newton to enter the break loop.* Thus, in the previous stack trace, you can see that the `viewClickScript` called `buttonClickScript` called `MethodB` called `MethodC` called `MethodD`.

Each frame in the stack trace contains five slots. The slots are:

`class`	Always `StackFrameInfo`. Not useful.
`CodeBlock`	The name of the function.
`programCounter`	The current point of execution within the function. Not useful since you have no way to go from program counter to source code line.
`receiver`	Always `NIL`. Not useful.
`contextFrame`	The names of the parameters and local variables in the function. You can obtain their values using `GetLocalFromStack`.

Note: If you enter a break loop because you've called a function with the wrong number of arguments, the third entry in the `StackTrace` array will correspond to the function you are trying to call. Unfortunately, it won't display the name of the function in the `CodeBlock` slot.

Displaying Local Variables and Parameters

```
GetLocalFromStack(stacklevel, 'variablename)
```

To display a local variable or parameter from one of the functions in the stack trace, use the global function `GetLocalFromStack`. The function takes two parameters. The first is the stack level, which is the array index in the stack trace array. The second parameter is the symbol of the variable whose value you want.

Thus, to get the value of `local1InB` from the `MethodB` function in the example stack trace, you would use:

```
GetLocalFromStack(4, 'local1InB);
```

There is also a useful shortcut for variables and parameters in the current function. Their values can be directly accessed using `Print`:

```
Print(local1InD);
```

Displaying self

```
GetSelfFromStack(level);
```

The value of `self` may change for different functions in the stack. To obtain the value of `self` for a given function, use `GetSelfFromStack`. The function takes one parameter, the stack level. As an example, to get the value of `self` for `MethodB` in the example stack trace described previously, use:

```
GetSelfFromStack(4);
```

As a shortcut for the current function, you can also use the variable `self`.

The Debugging Process

This section defines a number of techniques that can make debugging easier.

Use breakOnThrows

Normally, you will keep this variable set to `true`. Then, if an exception occurs, you don't have to repeat the process to find the error. Otherwise, when an error occurs, you would have to set `breakOnThrows` to `true`, and then redo the action before having access to the break loop debugging facilities.

Read Error Messages

While it might sound obvious, one of the most helpful things you can do is to read the error message in the Inspector carefully (the temptation is to rush right ahead with a stack trace).

When an exception occurs and you have `breakOnThrows` set, the Inspector will display a description of the error, and possibly some additional information. For example, the Inspector might display:

```
Exception |evt.ex.fr.type;type.ref.frame|: [-48404] Expected a number. Got:
{value: NIL}
```

In the above error, you can see that a number was expected. Most likely, the error occurred during an arithmetic operation (the most obvious operations requiring numbers). The other piece of information is that instead of a number it got a `NIL` value. Thus, the code that was executing used a `NIL` value instead of a number. At this point you might consider trying to track down which variables are set to `NIL` (perhaps with the prudent use of printing variable values).

In a similar vein, the Inspector might display:

```
Exception |evt.ex.fr.type;type.ref.frame|: [-48404] Expected a number. Got:
{value: "hello"}
```

Here, the same error has occurred, but the value is now `"hello"` rather than the desired integer.

Here's a final example:

```
Exception |evt.ex.fr.intrp;type.ref.frame|: [-48809] Undefined method "Foo"
{Symbol: Foo}
```

In this case, the code tried to call a method, `Foo`, which didn't exist.

Determine Where the Error Occurred

Knowing in which function the error occurred is pretty useful. To find out, use `StackTrace` from within the break loop. If you have more than one function with the same name (in different views), you'll need to figure out which one was executing by looking at the whole stack of functions. You can then reason from the structure of the program, look at the current value of `self`, or if all else fails, rename them uniquely.

Print Is Your Friend

If you have a complex function and you don't know where the error occurred, use some `Print` statements. If you sprinkle `Print` statements throughout the function, then you will know exactly where you choked—after the last `Print` that executed and before any of the others.

Remove Debug Code for Non-Debug Builds

You shouldn't distribute applications that contain `Print` statements. They slow your program down (slightly), they make your program larger (slightly), and programmers running your application with the Inspector open get (slightly) annoyed by the extraneous output that spews forth.

Rely on the `debugOn` constant. It is `true` when you are doing a debug build, and `NIL` otherwise (see "Debug Build" on page 378 for information on how to do a debug build). Having done so, you can restrict your `Print` statements to debug builds by using code like this:

```
if debugOn then
   Print("got to point A in the program");
non-debugging code
```

The `Print` statement will execute for a debug build, but will be stripped out of the compiled code for a non-debug build.

The NewtonScript compiler doesn't currently evaluate constant expressions at compile time, so don't use `debugOn` within other expressions like:

```
if debugOn and kExtremeDebuggingOn then
   Print("got to point A in the program");
non-debugging code
```

Although your debugging code won't execute in a non-debug build, it will still be part application.

Summary

In this chapter, we covered a wide variety of tools and techniques that you can use in debugging your applications. By now you should realize how useful a tool the Inspector is and have some idea what steps you would use to track down problems in your applications. You should also know better than to turn tracing on without having a plan for how to turn it off.

Appendix A

Important Methods

This appendix provides a brief description of the various view methods you may need to override. It is broken into two sections: the first contains methods described in this book, and the second contains the rest. Even though they are not used here they are useful for reference. Within each section, the methods are listed alphabetically.

Methods Covered in This Book

buttonClickScript()

This is called when a `protoTextButton`, `protoPictureButton`, `protoCloseBox`, `protoCancelButton`, or `protoPictRadioButton` is clicked on. It will also be called for any view that calls `TrackButton`.

For a `protoCloseBox` or a `protoCancelButton`, make sure to call `inherited:buttonClickScript()` so that the standard action takes place.

labelActionScript(*itemIndex*)

This routine is called for a `protoLabelInputLine` or `protoLabelPicker` when the user selects an item from the picker. *`itemIndex`* is the index of the chosen item.

For a `protoLabelInputLine`, if this method returns `true`, it signifies that it has handled it. If it returns `NIL`, the default action (setting the input field to the chosen item) is taken.

For a `protoLabelPicker`, the return result is ignored.

soupChanged(*soupName*)

When a soup changes, the `BroadcastSoupChange` global function should be called with the name of the changed soup. This function sends the `soupChanged` message to each application that registered with that soup name in the `soupNotify` global array (an application registers with two entries: its application symbol and a soup name). Other applications may call `BroadcastSoupChange`, or the system will call it when a card is inserted or removed.

textChanged()

This routine is called for a `protoLabelPicker` or `protoLabelInputLine` when the value of the `text` slot is changed.

textSetup()

This routine is called for a `protolabelInputLine` or a `protoLabelPicker` when it is instantiated. It should return a string to be used as the initial value of the input field.

viewChangedScript(*slotSymbol, view*)

When `SetValue` is called to change a view's slot value, it first changes the value and then sends the view the `viewChangedScript` message. This message contains the symbol of the slot, along with the view that was changed.

Override this message to be notified when slots in a view are changed (as long as they are changed by `SetValue`). The *`view`* parameter specifies the view that changed. It will usually, but need not always, equal `self`.

viewClickScript(*unit*)

This is called when a view is tapped on. In order to receive this message, the `vClickable` view flag must be set. *`unit`* contains information about the stroke. To turn off ink, call `InkOff(`*`unit`*`)`.

This method should return `true` if it handled the tap, and then no further processing of the tap will be done (strokes, gestures, words). If it returns `NIL`, the closest ancestor that is `vClickable` will be sent the same message. If no such ancestor is found, the default action is taken, which may include calls to `viewStrokeScript`, `viewGestureScript`, or `viewWordScript`.

viewHideScript()

The view system sends this message to a view when that view is closed or hidden. Subviews of a view which are closed or hidden do not receive the `viewHideScript` message.

viewOverviewScript()

This message is sent when the user taps the overview button. The message is sent to the frontmost view whose `viewFlags` has the `vApplication` bit set (normally your application's base view).

viewQuitScript()

The view system sends this message to a view when that view, or any ancestor view, is closed. When you send the `Quit` message to a view, that view will receive the `viewQuitScript` message before any descendants. However, the order in which descendants receive `viewQuitScript` messages is undefined.

viewScrollDownScript()

This message is sent when the user taps the down arrow. The message is sent to the frontmost view whose `viewFlags` has the `vApplication` bit set (normally your application's base view).

viewScrollUpScript()

This message is sent when the user taps the up arrow. The message is sent to the frontmost view whose `viewFlags` has the `vApplication` bit set (normally your application's base view).

viewSetupChildrenScript()

This message is sent just before any children of this view are created. Override this message to modify the `stepChildren` array when you want to create child views dynamically.

viewSetupDoneScript()

This message is sent after any children of this view are created, but before the view is drawn.

viewSetupFormScript()

This message is sent before the graphical representation of the view (viewCObject) is created. Override this message to set the size of the view dynamically.

viewShowScript()

The view system sends this message to a view when that view is opened or shown. Subviews of that view do not receive the `viewShowScript` message.

Methods Not Covered in This Book

buttonPressedScript()

This is called repeatedly while a `protoTextButton`, `protoPictureButton`, `protoCloseBox`, `protoCancelButton`, or `protoPictRadioButton` is pressed. Will also be called for a view that calls `TrackButton`.

changedSlider()

This is a required method for `protoSliders`. It is called whenever the user moves the gauge and lifts the pen. The current gauge setting is in the `viewValue` slot.

clusterChanged(*whichRadioButton*)

This is for a `protoRadioCluster`. This routine is called when a different radio button is turned on. The `buttonValue` slot of the newly turned-on radio button is passed as the *`whichRadioButton`* parameter.

DateFind(*findTime, filter, results, scope, statusContext*)

This is a method of the application's base view. It is called when the user does a "dates before" or "dates after" find. If an application doesn't have a `DateFind` method, a local Find will generate a "Date find not supported" slip. This method may be called when your application is closed.

The parameters are:

`findTime`	Time to look for in number of minutes since January 1, 1904.
`filter`	Either `'dateBefore` or `'dateAfter`.
`results`	Array of frames, one for each successful find. Append a frame to this frame if your application finds at least one item.
`scope`	Either `'localFind` or `'globalFind`.
`statusContext`	The Find view to which you can send status messages with the `SetStatus(`*`statusString`*`)` message.

FilingChanged()

This is called when the user files an item in a different folder. By the time this routine executes, the `labels` slot of the item has been modified and the entry has been saved to the soup.

FilterChanged()

This is called when the user selects a new folder from the folder tab. By the time this routine executes, the `labelsFilter` slot has already been changed.

Find(*what, results, scope, statusContext*)

This is a method of the application's base view. It is called when the user selects a "Find". If an application doesn't have a `Find` method, a local Find will generate a "Find not supported" slip.

The parameters are:

`what`	The string to Find.
`results`	Array of frames, one for each successful find. Append a frame to this frame if your application finds at least one item.
`scope`	Either `'localFind` or `'globalFind`.
`statusContext`	The Find view to which you can send status messages with the `SetStatus(`*`statusString`*`)` message.

This method may be called when your application is closed.

FindSoupExcerpt(*entry, finder*)

This is called to return the string displayed in the Find results slip for *`entry`*.

FlushEdits()

This routine is called for a `protoExpandoShell` when an expanded edit field is closed, and that edit field had been modified.

FolderChanged(*soupName, oldLabel, newLabel*)

This application base view method is called when the user deletes or renames a folder. It is sent to all applications in the `soupNotify` global array. The routine

should update all entries in the *soupName* soup—the `labels` slot value needs to be changed from *oldLabel* to *newLabel*. It should only do this if the application is responsible for that soup.

This method may be called when your application is closed.

keyPressScript(*keyResult*)

This routine is called for a `clKeyboardView` when the user presses a key. The default (if this method isn't present) converts *keyResult* into a sequence of characters that are posted as key events to the key receiver view.

labelClick(*unit*)

This routine is called for a `protoLabelInputLine` when the user clicks on the label. The *unit* parameter is the same as is passed to `viewClickScript`. If this method returns `true`, it signifies that it has handled the click. If it returns `NIL`, the default action (displaying a picker and then calling `labelActionScript`) is taken.

monthChangedScript()

This method is called for a `clMonthView` when the user changes the day.

pickActionScript(*itemNumber*)

This is used for a `clPickView`. The `pickActionScript` should be a method in the view located in the `callbackContext` slot of the pick view. *itemIndex* is the index of the picked item in the `pickItems` array.

pickerSetup()

This method is called for a `protoLabelPicker` when the user taps on the label. If it returns `true`, it signifies it has handled the tap. If it returns `NIL`, the default action (displaying a picker, setting the text field, and then calling the `labelActionScript`) is taken.

powerOffScript(*what*)

This method is called for a view that has registered itself as a power-off handler with `AddPowerOffHandler`. If *`what`* is `'okToPowerOff`, the method should return `true` to signify that it is okay to power off, or `NIL` to abort the power off. If *`what`* is `'powerOff`, the Newton is about to power off.

printNextPageScript()

This is called at the end of each page while printing or faxing to prepare the next page. It should return `NIL` if there are no more pages to print or fax.

PutAway(*item*)

This is a method of the application's base view. It is a message that is called to handle a frame that has been beamed from another Newton, store it, and redraw it if necessary. *`item`*`.body` contains the frame to store.

This method may be called when your application is closed.

SetupRoutingSlip(*fields*)

This is called to prepare a title and embed a frame for beaming or mailing. Set *`fields`*`.text` to the title, and *`fields`*`.target` to the frame.

ShowFoundItem(*entry, finder*)

This is called when the user taps on an item in the Find results slip (or if the user does a find that finds exactly one item). This routine should display the found item.

By the time this routine executes, the application is already open. If the result found in the results array (see discussion of `DateFind` or `Find`) protos from `ROM_compatibleFinder`, the second parameter is not present.

trackSlider()

This routine is called for a `protoGauge` whenever the `viewValue` slot changes. In particular, it is called repeatedly while the user moves the gauge knob.

valueChanged()

This routine is called when the value of a `protoCheckbox` changes.

viewAddChildScript(*newChildView*)

This method is called for clEditViews when *`newChildView`* is about to be added as a child. The method can modify *`newChildView`* before it is added. If the routine returns `true`, it signifies it handled adding the child. If it returns `NIL`, the default action (adding the view) is taken.

Caution: At the time this book was written, if this method returns `true` it must: add *`newChildView`* as a child, or it must add a new view which protos from *`newChildView`*.

---

viewDropChildScript(*childView*)

This method is called for clEditViews when *`childView`* is about to be deleted. If the routine returns `true`, it signifies that it removed the child from the `viewChildren` or `stepChildren` array. If it returns `NIL`, the default action (removing the child from the `viewChildren` or `stepChildren` array) is taken.

viewDrawScript()

This is called just after a view is drawn. Augment the standard view drawing by overriding this message.

viewGestureScript(*unit, gestureCode*)

If gesture recognition is turned on, the `viewGestureScript` will be called for gestures that the primitive view class does not handle. For example, a `clEditView` handles double tap, scrub, hilite, caret, and line gestures. On the other hand, a `clView` handles no gestures. Therefore, a scrub gesture in a `clEditView` will not cause its `viewGestureScript` to be called, whereas the same gesture in a `clView` does cause its `viewGestureScript` to be called.

The *`unit`* parameter is the same stroke unit passed to `viewClickScript`. The *`gestureCode`* parameter is 49 (tap), 50 (double-tap), 13 (scrub), 47 (hilite), 15(caret), 16 (line).

viewIdleScript()

This message is sent for views that have set up idling. To set up idling for a view, call `view:SetupIdle(milliseconds)` where *milliseconds* is the number of milliseconds before the next idle call. This routine can do whatever processing it desires, but should return the number of milliseconds before the next call to `viewIdleScript`. Returning `NIL` stops idling.

viewStrokeScript(*unit*)

If `vStrokesAllowed` is set, this method gets called when the user lifts the pen after a pen tap (for instance, when the user lifts the pen after writing a word in cursive). It is not called, however, if the `viewClickScript` returns `true`, since that implies the `viewClickScript` handled all pen interaction. In addition, it is only called for the first stroke of a multistroke letter or word.

This method should return `true` if it handled the tap; no further processing of the tap will be done (gestures, words). If it returns `NIL`, the closest ancestor that has `vStrokesAllowed` will be sent the same message. If no such ancestor can be found, the default action is taken, which may include calls to `viewGestureScript` or `viewWordScript`.

The *unit* parameter is the same as in the `viewClickScript` method.

viewWordScript(*unit*)

If word recognition is on, the `viewWordScript` will be called in cases where the primitive view class does not already handle words. For example, a `clEditView` handles words by entering them into a child `clParagraphView`. On the other hand, a `clView` does nothing with words. Therefore, entering a word in a `clEditView` will not cause its `viewWordScript` to be called, but entering a word in a `clView` will cause its `viewWordScript` to be called.

This method should return `true` if it handled the word. If it returns `NIL`, the closest ancestor that has word recognition enabled will be sent the same message. If there is no such ancestor, the default action is taken, which may include calls to `viewGestureScript` or `viewWordScript`.

The *unit* parameter is the same stroke unit passed to `viewClickScript`. `GetWordsArray(unit)` returns an array of words. The first entry is the most likely word. Subsequent entries are less probable possibilities for the word.

Appendix B

Important Messages

This appendix discusses many of the important messages that you will use in your applications. You will send messages to view, proto, and data objects.

View/Proto Messages

view:ChildViewFrames()

This returns the child views of `view` in an array. If `view` has no children, this function returns `NIL`.

view:Close()

This closes `view` and all of `view`'s descendants. The `viewQuitScript` of each child is called though the order of the calls for the descendants is undefined. The `viewCObject` for each view is destroyed and the views themselves are no longer referenced by the view system (and thus available for garbage collection). If a view is still referenced elsewhere (for instance, declared to some open ancestor), the view is not destroyed.

view:CopyBits(*picture, x, y, mode*)

Draws the bitmap object `picture` with its top left at the coordinates (*x*, *y*). The drawing mode is one of the following transfer modes: `modeCopy` (or equivalently, `NIL`), `modeOr`, `modeXor`, `modeBic`, `modeNotCopy`, `modeNotOr`, `modeNotXor`, `modeNotBic`.

view:Delete(*deleteMethodSymbol, parameterArray*)

This method gives you the visual effect of crumpling `view` and throwing it into a trash can. Underneath the crumpling view, the contents of a new view show simultaneously.

This method works by allocating a new offscreen bitmap. The current screen contents are saved in this new bitmap. Then, `Delete` locks the screen and sends the `deleteMethodSymbol` message to `view`, passing as parameters each of the elements of `parameterArray`. `deleteMethodSymbol` should either directly draw, or cause drawing to occur (for example, by calling `Dirty` or `SetValue`). At this point, `Delete` has a bitmap containing the old contents of `view`, and another containing the new contents of `view`. Animation occurs using those two bitmaps. At the conclusion, you are provided with the screen displaying the new contents of `view`.

rootView:Dial(*numberString, where*)

This method dials the number specified in `numberString` (which can contain numbers, the letters '`A`'-'`D`', '`*`', '`#`', '`,`', or '`-`'). The parameter, `where`, is either `'speaker` or `'modem`.

view:Dirty()

This method marks `view` as dirty and in need of redrawing. It is redrawn at the next idle time (when there are no other pending events).

view:DoDrawing(*drawMethodSymbol, parameterArray*)

You will call this message when you want to draw somewhere other than within your `viewDrawScript`. This routine sets up clipping and then allows you to do your drawing by sending the `drawMethodSymbol` message to `view`. It uses the parameters specified in `parameterArray`. Afterward, it restores the clipping.

view:Drag(unit, bounds)

This method drags `view` as the user moves the stylus on the screen. It is called from within a `viewClickScript`. The `unit` parameter is the same one passed to `viewClickScript`. The `bounds` is a bounding box (in global coordinates) limiting the dragging of `view`. If `bounds` is `NIL`, `view` can be dragged anywhere on screen.

view:DrawShape(*shape, styleFrame*)

This message draws in `view`. The `shape` parameter is either a shape (line, rectangle, rounded-rectangle, oval, arc, polygon, region, picture, or text) or an array of shapes and style frames. The `styleFrame` parameter is a frame specifying a style to use for drawing `shape`. The slots in `styleFrame` are: `transferMode`, `penSize`, `penPattern`, `fillPattern`, `font`, and `justification`.

view:Effect(*effect, reveal, sound, messageSymbol, parameterArray*)

This method does a visual animation effect on `view` as specified by the effect parameter. (These are the same effects as in `viewEffect`.) It also plays the sound specified by `sound` if it is not `NIL`.

This method works by allocating a new offscreen bitmap. The current screen contents are saved in this new bitmap. Then, `Effect` locks the screen and sends the `messageSymbol` message to `view`, passing as parameters each of the elements of `parameterArray`. `messageSymbol` should directly draw or cause drawing to occur (for example, by calling `Dirty` or `SetValue`). At this point,

`Effect` has a bitmap containing the old contents of *`view`* and another containing the new contents of *`view`*. Animation occurs using those two bitmaps and you end up with the screen displaying the new contents of *`view`*. If *`reveal`* is non-`NIL`, then `Effect` uses a reveal line at the edge between the old contents and the new contents.

protoPicker:GetItemMark(*itemIndex*)

This returns the mark associated with the item in the picker at *`itemIndex`*. It returns `NIL` if there is no mark associated with that item.

view:GlobalBox()

This method returns the bounds of *`view`* in global screen coordinates. It returns a frame with `left`, `right`, `top`, and `bottom` slots.

view:Hide()

This method hides *`view`*, so that it no longer displays on screen. The `viewCObject` still exists, however. It also sends *`view`* the `viewHideScript` message.

view:Hilite(*hiliteOn*)

If *`hiliteOn`* is non-`NIL`, this method highlights *`view`* (according to the highlight specified in `viewFormat`). If *`hiliteOn`* is `NIL`, this method unhighlights *`view`*. You can use this as a toggling message.

view:HiliteUnique(*hiliteOn*)

If *`hiliteOn`* is `NIL`, this method unhighlights *`view`*. If *`hiliteOn`* is non-`NIL`, this method highlights *`view`* (according to the highlight specified in `viewFormat`). It also ensures that all siblings of *`view`* are unhighlighted.

view:LayoutColumn(childTemplateArray, childIndex)

This method is used to calculate a subset of child templates from *`childTemplateArray`* and display them in a column in *`view`*. The *`childIndex`* parame-

ter specifies the index of the first child to display from the *childTemplateArray*.

LayoutColumn returns an array of children from *childTemplateArray* that are within the bounds of *view*. You can use this array as a stepChildren array.

view:LayoutTable(tableDefinition, columnIndex, rowIndex)

This method creates a matrix of child templates from a larget matrix of templates. The *tableDefinition* frame describes the matrix of templates that will be shown in *view*; the *columnIndex* parameter specifies which column appears at the left of view; and the *rowIndex* parameter specifies which row appears at the top of the view.

The *tableDefinition* frame contains:

tabAcross	The number of columns in the matrix.
tabDown	The number of rows in the matrix.
tabWidths	The integer width of the columns or an array of integer widths.
tabHeights	The integer height of the rows or an array of integer heights.
tabProtos	The template to be used for each matrix element or an array of templates.
tabValueSlot	A symbol which stores the names of each matrix element view.
tabValues	The value to be stored in each matrix element, or an array of those values.
tabSetup	A method (func(*childTemplate, columnNumber, rowNumber*)) that is called before each matrix element view is created.

For any slot that can take an array, the elements of the array are mapped down the first column of the matrix, then down the second column, and so on. If there aren't enough elements in the array, the mapping starts over with the array again.

view:LocalBox()

This method returns the bounds of `view` in coordinates local to the view. It returns a frame with `left`, `right`, `top`, and `bottom` slots. The `left` and `top` slots will always be `0`. The `right` and `bottom` slots contain the width and height of `view`.

view:LockScreen(*lock*)

All drawing is done to an offscreen bitmap and then copied from off screen to on screen. Before a view (or hierarchy of views) draws, the screen is locked so that the copy from off screen to on screen does not take place until the drawing is completed. When all the drawing is completed, the screen is unlocked and updated. The visual effect is of a single update, rather than a succession of views with drawing in bits and pieces.

If you draw within your `viewDrawScript`, the screen has already been locked. If you draw from within any other method, you should call `:LockScreen(true)` to lock the screen before drawing, and then call `:LockScreen(NIL)` after drawing to unlock it.

rootView:Notify(*level, headerStr, messageStr*)

This method notifies the user of an event. The `headerStr` is a header at the top of the notification slip. The `messageStr` is the actual message (which should be `EnsureInternal`ed in case the user scrolls up to a message from an application which is no longer present). The `level` parameter specifies the urgency of the message. It is one of the following:

`kNotifyLog`	The user is not formally alerted, but the error is put in the notification log. Users can see the message if another notification occurs and they scroll back through the notification slip.
`kNotifyMessage`	An icon blinks onscreen until the user taps it. The notification slip is displayed then.
`kNotifyQAlert`	The notification slip is displayed immediately.
`kNotifyAlert`	The notification slip is displayed immediately, and the system beep is played.

view:Open()

This method opens `view` and all its descendants (unless they are not `vVisible`). It sends the `viewSetupFormScript`, `viewSetupChildrenScript`, `viewSetupDoneScript` messages to `view` and all its descendants. It then sends the `viewShowScript` message to `view`.

view:Parent()

This method returns the parent of `view`. Use this method rather than directly accessing the `_parent` slot of `view`.

view:RedoChildren()

This method destroys all the children of `view`, sends `view` the `viewSetupChildrenScript` message, and then recreates them. Consider using `SyncChildren` instead, as it does not destroy and recreate child views that remain inside the bounds of `view`.

view:RevealEffect(*numPixels, bounds, sound, methodSymbol, parameterArray*)

This method does a reveal animation effect on the portion of `view` specified by `bounds`. It also plays the sound specified by `sound` (unless it is `NIL`). It moves the specified onscreen portion of `view` by `numPixels` pixels (a positive number moves the bits up, a negative number moves them down). This method doesn't affect `view` itself, but only the bits on screen.

This method words by allocating a new offscreen bitmap. The current screen contents are saved in this new bitmap. Then, `RevealEffect` locks the screen and sends the `messageSymbol` message to `view`, passing as parameters each of the elements of `parameterArray`. `messageSymbol` should either directly draw or cause drawing to occur (for example, by calling `Dirty` or `SetValue`). At this point, `RevealEffect` has a bitmap containing the old contents of `view` and another containing the new contents of `view`. It then does animation using those two bitmaps, ending up with the screen displaying the new contents of `view`.

protoRadioCluster:SetClusterValue(whichButton)

This method changes the selected radio button to the one whose `buttonValue` slot is equal to `whichButton`. It updates the radio buttons on screen, sends itself

the `ClusterChanged` message, and is undoable (that is, the user can undo the change in radio button).

clParagraphView:SetHilite(start, end)

This method highlights the characters in *`clParagraphView`* in the range *`start`* to *`end`*. Before calling this method, the view must have been set as a key view using `SetKeyView`.

protoPicker:SetItemMark(*itemIndex, markCharacter*)

This method sets the mark for the item at *`itemIndex`* to *`markCharacter`*.

protoLabelInputLine:SetLabelCommands(*labelArray*)

This updates the `labelCommands` slot to the array of strings in *`labelArray`*.

protoLabelInputLine:SetLabelText(*label*)

This updates the label to the string in *`label`*.

view:SetOrigin(originX, originY)

This method changes the origin of *`view`* to (*`originX`*, *`originY`*). It dirties *`view`* so that it is redrawn. Likewise, *`view`*'s children are redrawn relative to the new origin. This method is one way to implement scrolling.

view:SetPopup()

This method causes *`view`* to be a popup view. A popup view is closed on the next pen tap.

view:SetupIdle(*delay*)

This method sets up idling for *`view`*. The `viewIdleScript` message will be sent to *`view`* after the number of milliseconds specified in *`delay`*.

view:Show()

This message shows `view` on screen and sends it the `viewShowScript` message. If `view` is not already open, `Show` opens it.

view:SlideEffect(*amtToScroll, amtToMove, sound, methodSymobl, parameterArray*)

This method does a scrolling animation effect on `view`. It also plays the sound specified by `sound` (unless it is `NIL`). The sound parameter is commonly one of `scrollUpSound` or `scrollDownSound`. It scrolls the bits on screen within the bounds of `view` by `amtToScroll` pixels (a positive number moves the bits down, a negative number moves them up). It moves the entire view on screen by `amtToMove` pixels (a positive number moves the bits up, a negative number moves them down). This method does not affect `view` itself, but only the bits on screen.

This method words by allocating a new offscreen bitmap. The current screen contents are saved in this new bitmap. Then, `SlideEffect` locks the screen and sends the `messageSymbol` message to `view`, passing as parameters each of the elements of `parameterArray`. `messageSymbol` should either directly draw or cause drawing to occur (for example, by calling `Dirty` or `SetValue`). At this point, `SlideEffect` has a bitmap containing the old contents of `view` and another containing the new contents of `view`. It then does animation using those two bitmaps, ending up with the screen displaying the new contents of `view`.

view:SyncChildren()

This method redraws all the children of `view` , creating children that are now within the bounds and deleting any child views that are no longer within the bounds of `view`.

First, it sends `view` the `viewSetupChildrenScript` message and then creates and deletes necessary children. If any children have moved, they are sent the `SyncView` message.

view:SyncScroll(childViewArray, childIndex, upOrDown)

This method scrolls the children in `childViewArray` up the height of the top child (if `upOrDown` is `-1`), or down the height of the bottom child (if `upOrDown` is `1`). The `childIndex` parameter is the index within `childViewArray` of the top child within `view`.

The method plays the `scrollUpSound` or `scrollDownSound` as appropriate and returns the array of child views that are visible within *`view`* after scrolling. `SyncScroll` looks for the optional slots, `allCollapsed` and `collapsedHeight`, in view. It looks for the optional slot `collapsed` and the required slot `height` in each child view.

The optional slots are used for views that support children and that have both expanded and collapsed modes. The slots values are:

`allCollapsed`	Contains `true` if all children are collapsed; `NIL` otherwise.
`collapsedHeight`	Contains the height of child views in the collapsed state.
`collapsed`	Contains `true` if this child view is collapsed; `NIL` otherwise.
`height`	Contains the height in pixels of this child view in the expanded state.

view:SyncView()

This method updates the `viewCObject` to reflect changes in *`view`* slot values (for example, the `viewBounds`). It sends the `viewSetupFormScript` message to *`view`* before updating the `viewCObject`.

If you use the `SetValue` function to set `viewBounds`, `viewFlags`, `viewFormat`, `viewJustify`, or `viewFont`, you need not explicitly call `SyncView`. If you change the values of one of these slots directly, you must call `SyncView` before that view reflects the changes.

rootView:SysBeep()

This method plays the system beep sound (or flashes the screen if the volume is off).

view:Toggle()

If *`view`* is open, this method sends it the `Close` message. If view is closed, the reverse happens and this method sends the `Open` message.

protoCheckbox:ToggleCheck()

This method toggles the state of its checkmark. The `viewValue` slot holds the current state of the checkmark: `true` if on, `NIL` if off.

view:TrackButton(*unit*)

This method is called from within a `viewClickScript`. The `unit` parameter is the same one passed to `viewClickScript`. The method makes a click sound and then starts tracking the pen. While the pen is in the bounds of view, *`view`* is highlighted; if the pen goes outside the bounds, *`view`* is unhighlighted. While the pen is in the bounds, `TrackButton` repeatedly sends view the `buttonPressedScript` message (if it exists). If the pen is released while within the bounds of *`view`*, this method sends view the `buttonClickScript` message. In any case, `TrackButton` unhighlights *`view`* before returning.

If an error occurs while tracking (if the *`buttonPressedScript`* method causes an error), this method will unhighlight *`view`*.

view:TrackHilite(*unit*)

This method is called from within a `viewClickScript`. The `unit` parameter is the same as that passed to `viewClickScript`. The method makes a click sound and then starts tracking the pen. While the pen is in the bounds of view, *`view`* is highlighted; if the pen goes outside the bounds, *`view`* is unhighlighted. While the pen is in the bounds, `TrackHilite` repeatedly sends *`view`* the `buttonPressedScript` message (if it exists). `TrackHilite` returns `true` if the pen is released while within the bounds of *`view`* (and leaves view highlighted); it returns `NIL` otherwise.

protoFilingButton:Update()

This method updates the filing button to display a triangle if the current target is on an external store. It erases the triangle if the target is `NIL` or if the target is not on an external store.

protoLabelPicker:UpdateText(*newText*)

This method changes the currently selected item to *`newText`*.

protoLabelInputLine:UpdateText(*newText*)

This method changes the text in the input line to `newText`. It is undoable (the user can undo the change).

protoShowBar:UpdateFilter(*oldFolder, newFolder*)

This method updates the show bar to display a new folder. You need to call this if you change the folder programmatically (rather than allowing the user to change it). The `oldFolder` and `newFolder` parameters are symbols describing folders.

Store Methods

store:CreateSoup(*soupName, arrayOfIndexes*)

This store method creates a soup named `soupName` on `store`. The second parameter is an array of frames. Each frame in that array specifies a soup index and has the slots `structure` (with the value `'slot`), `path` (containing a symbol or path expression), and `type` (one of `'int`, `'string`, `'char`, `'real`, or `'symbol`).

store:Erase()

This completely erases `store`.

store:GetKind()

This method returns the media type of `store`. The possibilities (for Newtons localized for the U.S.) are `"Internal"`, `"Flash storage card"`, `"Storage card"` and `"Application card"`.

store:GetName()

This method returns the name of `store`.

store:GetSignature()

This method returns the integer signature of `store`.

store:GetSoup(*soupName*)

This method gets the soup named *`soupName`* from `store`. If no such soup exists, the method returns `NIL`.

store:GetSoupNames()

This method returns an array containing the names of all the soups on a specified `store`.

store:HasSoup(*soupName*)

This method returns `true` if `store` contains a soup named `soupName`. It returns `NIL` otherwise.

store:IsReadOnly()

This returns `true` if `store` is in a read-only state. This returns `NIL` if `store` can be written.

store:SetName(*newName*)

This sets the name of `store` to `newName` (if the store is writable).

store:SetSignature(*newSignature*)

This method sets the integer signature of `store`.

store:TotalSize()

This returns the total size of `store` in bytes.

store:UsedSize()

This returns the number of bytes currently used in `store`.

Soup Methods

unionSoup:AddToDefaultStore(*frame*)

This method takes `frame` and adds it to `unionSoup`. The entry frame is placed on the default store. The user specifies the default store in the Card slip. An error will be generated if the default store does not contain that soup.

soup:CopyEntries(*destSoup*)

This method copies all the entries from `soup` to `destSoup`. Neither `soup` nor `destSoup` may be a union soup.

soup:GetIndexes()

This method returns an array of frames, one for each index in `soup`. This method is not supported for union soups.

soup:GetNextUid()

This returns the value of the `_uniqueID` that will be assigned to the next entry added to `soup`. This method is not supported for union soups.

soup:GetSignature()

This method returns the signature of `soup` and is not supported for union soups.

soup:RemoveIndex(*indexPath*)

You use this to remove a `soup` index on the path `indexPath`. This method works on both union and nonunion soups.

soup:SetSignature(*signature*)

When a soup is created, it is assigned a random integer as a signature. With this signature, you can tell whether a soup has been removed and a new soup added with the same name. This method sets the signature of `soup` to `newSignature` and is not supported for union soups.

Cursor Methods

cursor:Clone()

The `Clone` method returns a new cursor that shares the same query as `cursor`. The new cursor has an independent location and, as such, provides you with a way to have multiple iterators over the same entries.

cursor:Entry()

This returns the current entry from `cursor`. If the entry gets deleted while the cursor is still on it, `Entry` returns `'deleted` and no longer rests on a soup entry.

cursor:Goto(*entry*)

This function moves `cursor` to `entry`. The `entry` must be an entry in the soup and have already been returned by a call to `Entry`, `Next`, or `Prev`.

cursor:GotoKey(*keyValue*)

This moves `cursor` to the first entry with an indexed value equal to `keyValue`. If no such entry is present, the cursor is put on the next indexed entry. This method should only be called for queries that have specified an `indexPath`.

cursor:Move(*numEntries*)

This moves `cursor` forward or backward by `numEntries`. `Move(1)` is equivalent to `Next()` and `Move(-1)` is equivalent to `Prev()`. This function is faster than repeatedly calling `Next` or `Prev`.

cursor:Next()

This moves `cursor` to the next entry satisfying the query and then returns it. If there is no next entry, `Next` returns `NIL`.

cursor:Prev()

This moves `cursor` backward to the previous entry that satisfies the query and returns it. If you are at the beginning and there is no previous entry, `Prev` returns `NIL`.

cursor:Reset()

This moves `cursor` back to the first entry that satisfies the query.

Appendix C

Important Global Functions

This chapter covers the available global functions (each of which is a slot in `GetGlobals().functions`. Within each section, the functions are presented alphabetically.

Global Functions Covered in This Book

AddArraySlot(*array, value*)

This adds *`value`* as a new element in *`array`*. The new element is at the end of the array.

AddDeferredAction(*function, parameterArray*)

This calls `function` with the parameters in `parameterArray` the next time through the event loop. This provides a way to defer execution of code for a short period.

AddStepView(*view, childTemplate*)

This function creates a view from `childTemplate`, and adds it as a child view to `view`.

Array(*numberElements, initialValue*)

This function returns a new array with `numberElements` elements. Each element has the value `initialValue`.

ArrayPos(*array, value, startIndex, testFunction*)

This function searches for `value` in `array`, starting at `startIndex`. `testFunction` is a function that does equality testing on its two parameters. If `testFunction` is `NIL`, = is used for equality testing.

ArrayRemoveCount(*array, startIndex, count*)

This removes `count` elements from `array`, starting at `startIndex`.

Band(*integerA, integerB*)

This returns the bit-wise and of `integerA` and `integerB`.

Bnot(*integer*)

This returns the bit-wise complement of `integer`.

Bor(*integerA, integerB*)

This returns the bit-wise or of `integerA` and `integerB`.

Bxor(*integerA, integerB*)

This returns the bit-wise exclusive or of `integerA` and `integerB`.

BreakLoop()

This enters a break loop.

BroadcastSoupChange(*soupName*)

This notifies applications (those that are interested) of the change to `soupName`.

Clone(*value*)

This makes a duplicate of value, one-level deep.

DateNTime(*time*)

This converts time (number of minutes since January 1, 1904) to a string displaying both the date and time.

Debug(*value*)

This prints `value` to the inspector, followed by a newline. Strings are surrounded with ", and characters are preceded by `$`.

DeepClone(*value*)

This returns a recursive clone of `value`.

Display(*value*)

This prints `value` to the inspector. Strings are surrounded with " ", and characters are preceded by `$`.

DV(*view*)

This displays a one-line summary of `view` and each of its descendants in the Inspector. It also briefly flashes `view` on the Newton.

EnsureInternal(*value*)

If `value`, and everything within `value` (recursively), is in either internal memory or on the system ROM, then this just returns `value`. Otherwise, it `Clones` as necessary to return a copy of `value`. It ensures that the copy, and everything within it, are either in internal memory or on the system ROM.

EntryChange(*entry*)

This writes `entry` back into its soup and updates its modification time.

EntryCopy(*entry, newSoup*)

This copies `entry` from its current soup to `newSoup`.

EntryModTime(*entry*)

This returns the last modification time of `entry`. It returns `0` if `entry` has not been modified.

EntryMove(*entry, newSoup*)

This moves `entry` from its current soup to `newSoup`.

EntryRemoveFromSoup(*entry*)

This removes `entry` from its soup.

EntryReplace(*entry, newEntry*)

This replaces `entry` with the contents of `newEntry`, while retaining the original unique ID.

EntrySize(*entry*)

This returns the number of bytes `entry` occupies in its soup.

EntrySoup(*entry*)

This returns the soup to which `entry` belongs.

EntryStore(*entry*)

This returns the store of the soup to which `entry` belongs.

EntryTextSize(*entry*)

This returns the number of bytes in a soup that that the string slots of `entry` occupy.

EntryUndoChanges(*entry*)

This reads `entry` from its soup, wiping out any changes that have been made to `entry` since it was last read or written.

EntryUniqueID(*entry*)

This returns the unique ID of `entry`.

ExitBreakLoop()

This exits the current break loop.

FormattedNumberStr(*number, formatSpecString*)

This converts `number` to a string as specified by `formatSpecString`. `formatSpecString` is used like the format string of `printf` in C.

For example,

```
FormattedNumberStr(3.5, "%0.2f")
```

yields:

```
3.50
```

FrameDirty(*frame*)

This returns `true` if *`frame`* has been modified since it was last saved in a soup and returns `NIL` otherwise.

GC()

This causes an immediate garbage collection. Other than taking a little time, this function should have no effect on your application. The only objects garbage collected are those that can no longer be reached. You will rarely, if ever, need to call this routine.

GetAppParams()

This returns a frame with information about the Newton. Currently, the information includes the size of the screen and the location of the icon bar. More information may be provided in this frame in the future.

GetGlobals()

This returns the globals frame that contains one slot for each global variable. The frame includes a `functions` slot that contains an array of all global functions.

GetLocalFromStack(*symbol, stackLevel*)

While within a break loop, this returns the value of the local variable or parameter whose symbol is *`symbol`*. A *`stackLevel`* of 2 is the current function.

GetRoot()

This returns the root view. Within this view, there is a slot for each application. The slot symbol is the application symbol; the value is the application base view.

GetSelfFromStack(*stackLevel*)

While in a break loop, this returns the value of `self` associated with the specified *`stackLevel`*. A *`stackLevel`* of 2 is the current value of `self`.

GetSlot(*frame, slotSymbol*)

This returns the value of the slot whose symbol is `slotSymbol` in `frame`. It returns `NIL` if no such slot is found. This function does not use proto or parent inheritance.

GetStores()

This returns an array of stores. The first entry in the array is the internal store.

GetUnionSoup(*soupName*)

This returns a union soup composed of all soups with the name `soupName`.

GetVariable(*frame, slotSymbol*)

This returns the value of the slot whose symbol is `slotSymbol` in `frame`. It returns `NIL` if no such slot is found. This function uses both proto and parent inheritance.

GetView(*template*)

This returns the viewCObject associated with `template`. Returns `NIL` if the view is not open.

HasSlot(*frame, slotSymbol*)

This returns true if `frame` contains a slot whose symbol is `slotSymbol`. It returns `NIL` if no such slot is found. This function does not use proto or parent inheritance.

HasVariable(*frame, slotSymbol*)

This returns true if `frame` contains a slot whose symbol is `slotSymbol`. It returns `NIL` if no such slot is found. This function uses both proto and parent inheritance.

InkOff(*unit*)

This turns off the automatic ink that is drawn as the user moves the stylus. `unit` is the same parameter used by `viewClickScript`, `viewStrokeScript`, or `viewWordScript`.

Intern(*string*)

This returns the symbol that has the name `string`.

IsSoupEntry(*frame*)

This returns `true` if `frame` is an entry in a soup; it returns `NIL` otherwise.

Length(*array*)

This returns the number of elements in `array`. Do not use this for strings; use `StrLen` instead.

MapCursor(cursor, applyFunction)

This applies `applyFunction` to each entry in `cursor`. It returns an array with the (non-`NIL`) return results of `applyFunction`.

Min(*integerA, integerB*)

This returns the minimum of `integerA` or `integerB`.

NumberStr(*number*)

This converts `number` to a string.

Query(*soup, querySpec*)

This creates a cursor for `soup` based on the `querySpec`. Types of queries are index, words, and text.

Random(*lowerBoundInteger, upperBoundInteger*)

This returns a random integer in the range [`lowerBoundInteger`, `upperBoundInteger`].

RelBounds(*left, top, width, height*)

This returns a frame `{left: left, right: left + width, top: top, bottom: top + height}`.

RemoveSlot(*arrayOrFrame, arrayIndexOrSlotSymbol*)

This function removes a specified slot from a frame or element from an array.

RemoveStepView(*parentView, childView*)

This removes `childView` from `parentView` and closes it.

RIntToL(*realNumber*)

This rounds `realNumber` to its nearest integral value. The return result is an integer.

Round(*realNumber*)

This rounds `realNumber` to its nearest integral value. The return result is a real number with the fraction part equal to 0.

SetAdd(*array, value, uniqueFlag*)

If `uniqueFlag` is non-`NIL`, this function adds `value` to `array` when it is not already present. If `uniqueFlag` is `NIL`, the function adds `value` to the end of `array`.

SetContains(*array, value*)

This returns `true` if `value` is present in `array`; it returns `NIL` otherwise.

SetDifference(*arrayA, arrayB*)

This returns an array containing those elements that are in `arrayA` but are not in `arrayB`.

SetLength(*array, newLength*)

This sets the length of `array` to `newLength`. If the array has to grow, the new elements are `NIL`.

SetOverlaps(*arrayA, arrayB*)

This function returns the index of the first element of `arrayA` that is also present in `arrayB`. If no such element is found, it returns `NIL`.

SetRemove(*array, value*)

If `value` is present in `array`, this function removes it.

SetUnion(*arrayA, arrayB, uniqueFlag*)

This function returns an array with all the elements from `arrayA` and `arrayB`. If `uniqueFlag` is non-`NIL`, then duplicates are not added.

SetValue(*view, slotSymbol, value*)

This function does the following: sets the `slotSymbol` slot of `view` to `value`; sends the `viewChangedScript` message to the view; and redraws the view if an important slot has changed (such as `viewBounds`, `viewFlags`, `text`, and so on).

StrEqual(*stringA, stringB*)

This returns `true` if `stringA` contains the same characters as `stringB`, and `NIL` otherwise. This function ignores case.

StringToDate(*string*)

This converts `string` to a time (number of minutes since January 1, 1904). The function returns `NIL` if `string` doesn't represent a date.

If `string` contains a time, but no date, the current day is used. In a fit of foolish consistency, if string contains a date, but no time, the current time is used.

StringToNumber(*string*)

This converts `string` to a real number. The function returns `NIL` if `string` doesn't represent a number.

StrLen(*string*)

This returns the number of characters in `string`.

Time()

This returns the number of minutes since January 1, 1904, as an integer.

Visible(*view*)

This returns `true` if `view` is visible on screen (open and not hidden).

Write(*value*)

This prints `value` to the Inspector.

Global Functions Not Covered in This Book

`Abs(number)`

`Acos(number)`

`Acosh(number)`

`AddDeferredAction(function, parameterArray)`

AddDelayedAction(*function, parameterArray, delay*)

AddPowerOffHandler(*view*)

AddToUserDictionary(*wordString*)

AddUndoAction(*methodSymbol, parameterArray*)

AddWordToDictionary(*dictionary, wordString*)

Annuity(*ratePerPeriod, numberPeriods*)

Apply(*function, argumentArray*)

ArrayMunger(*dstArray, dstStart, dstCount, srcArray, srcStart, srcCount*)

ArrayToPoints(*pointsArray*)

Asin(*number*)

Asinh(*number*)

Atan(*number*)

Atan2(*numberX, numberY*)

Atanh(*number*)

BeginsWith(*string, subString*)

BuildContext(*template*)

Capitalize(*string*)

CapitalizeWords(*string*)

Ceiling(*real*)

CheckThatFolderExists(*frame, folderSymbol*)

Chr(*integer*)

ClassOf(*value*)

Compound(*ratePerPeriod, numPeriods*)

CopySign(*numberX, numberY*)

Cos(*number*)

Cosh(*number*)

Date(*time*)

DisposeDictionary(*dictionary*)

DoPopup(*list*, *left*, *top*, *notifyView*)

Downcase(*string*)

DrawXBitmap(*bounds*, *picture*, *index*, *drawingMode*)

EndsWith(*string*, *subString*)

EntryChangeWithModTime(*entry*)

EntryReplaceWithModTime(*entry*, *newEntry*)

Erf(*number*)

Erfc(*number*)

EvalStringer(*frame*, *array*)

Exp(*number*)

Expm1(*number*)

ExtractByte(*data*, *offset*)

ExtractBytes(*data*, *offset*, *length*, *class*)

ExtractChar(*data*, *offset*)

ExtractCString(*data*, *offset*)

ExtractLong(*data*, *offset*)

ExtractPString(*data*, *offset*)

ExtractWord(*data*, *offset*)

Fabs(*number*)

Fdim(*numberX*, *numberY*)

FeClearExcept(*number*)

FeGetEnv()

FeGetExcept(*number*)

FeGetRound()

FeHoldExcept(*number*)

```
FeRaiseExcept(number)
FeSetEnv(number)
FeSetExcept(numberX, numberY)
FeSetRound(number)
FeTestExcept(number)
FeUpdateEnv(number)
FindStringInArray(array, string)
FindStringInFrame(frame, stringArray, reportPaths)
Floor(real)
Fmax(numberX, numberY)
Fmin(numberX, numberY)
Fmod(number)
FontAscent(fontSpec)
FontDescent(fontSpec)
FontHeight(fontSpec)
FontLeading(fontSpec)
FormatVertical(bounds, justify)
Gamma(numberX)
GetCaretBox()
GetDictionaryData(dictionary)
GetPackages()
GetPoint(selector, unit)
GetPointsArray(unit)
GetRandomDictionaryWord(minLength, maxLength)
GetRandomWord(minLength, maxLength)
GetScoreArray(unit)
GetViewFlags(template)
```

```
GetVolume()
GetWordArray(unit)
HiliteOwner()
HitShape(shape, x, y)
HourMinute(time)
Hypot(numberX, numberY)
InkOn(unit)
Interpret(codeBlock, frame)
IsFinite(number)
IsNan(number)
IsNormal(number)
LdExp(numberX, numberY)
LGamma(number)
Log(number)
Log10(number)
Log1p(number)
Logb(number)
LongDateStr(time, dateStrSpec)
LookupWord(wordString)
MakeLine(x1, y1, x2, y2)
MakeOval(left, top, right, bottom)
MakePict(shapeArray, styleFrame)
MakePolygon(pointsArray)
MakeRect(left, top, right, bottom)
MakeRegion(shapeArray)
MakeRoundRect(left, top, right, bottom, diameter)
MakeShape(object)
```

```
MakeText(string, left, top, right, bottom)
MakeWedge(left, top, right, bottom, startAngle, arcAngle)
Max(integerA, integerB)
NearbyInt(number)
NextAfterD(numberX, numberY)
NewDictionary(dictionaryKind)
OffsetShape(shape, deltaH, deltaV)
Ord(character)
ParamStr(baseString, paramStrArray)
ParseUtter(string)
Perform(frame, messageSymbol, parameterArray)
PlaySound(soundFrame)
PlaySoundSync(soundFrame)
PointsToArray(polygonShape)
PostKeyString(view, keystrokeString)
Pow(numberX, numberY)
PrimClassOf(value)
PtInPicture(x, y, bitmap)
RandomX(numberX, numberY)
Real(integer)
RefreshViews()
RegTaskTemplate(taskTemplate)
Remainder(numberX, numberY)
RemovePackage(packageFrame)
RemovePowerOffHandler(view)
RemQuo(numberX, numberY)
ReplaceObject(originalBinaryObject, targetBinaryObject)
```

```
RInt(number)
Round(number)
SaveUserDictionary()
Scalb(numberX, numberY)
ScaleShape(shape, srcRect, dstRect)
SetBounds(left, top, right, bottom)
SetClass(arrayFrameOrBinaryObject, classSymbol)
SetDictionaryData(dictionary, binaryObject)
SetInkerPenSize(integerFrom1To4)
SetKeyView(view, characterOffset)
SetRandomSeed(integer)
SetTime(time)
SetVolume(integerFrom0To4)
ShapeBounds(shape)
ShortDate(time)
ShortDateStr(time, dateStrSpec)
Sign(number)
SignBit(number)
Sin(number)
Sinh(number)
Sort(array, test, key)
SplitString(string)
SPrintObject(value)
Sqrt(number)
Stats()
StrCompare(stringA, stringB)
StrConcat(stringA, stringB)
```

StrFontWidth(*string*, *fontSpec*)

Stringer(*array*)

StringToTime(*timeString*)

StrMunger(*dstString*, *dstStart*, *dstCount*, *srcString*, *srcStart*, *srcCount*)

StrokeBounds(*unit*)

StrokeDone(*unit*)

StrokeInPicture(*unit*, *bitmap*)

StrPos(*string*, *subString*, *startPosition*)

StrReplace(*string*, *subString*, *replacementString*, *count*)

StrTruncate(*string*, *lengthInPixels*)

StrWidth(*string*)

SubStr(*string*, *startPosition*, *numberCharacters*)

Tan(*number*)

Tanh(*number*)

Ticks()

TieViews(*mainView*, *dependentView*, *methodSymbol*)

TimeInSeconds()

TimeStr(*time*, *timeStrSpec*)

TotalMinutes(*dateFrame*)

TotalMinutes(*dateFrame*)

TrimString(*string*)

Trunc(*number*)

UnRegTaskTemplate(*templateID*)

Upcase(*string*)

Appendix D

Important Global Variables

This appendix discusses many of the important global variables (each of which is a slot in `GetGlobals()`).

Variables Covered in This Book

soupNotify

This is an array in which an application adds entries if it wants to be notified when a particular soup changes. The application registers in `soupNotify` by adding two entries to the array: the first is a particular soup name (a string) and the second is an application symbol. When `BroadcastSoupChange` is called

with a soup name, it sends the `soupChanged` message to each application that registered with that soup name.

functions

This frame contains one entry for each global function. The name of each slot is the name of a function, while the value is a code block.

printDepth

An integer containing the depth that print statements use. The default value is `1`.

trace

If `trace` is equal to `'functions`, all function calls are printed. If it is equal to `true`, all function calls and variable assignments are printed. The default value of `trace` is `NIL`.

Variables Not Covered in This Book

findApps

An array of application symbols that support global find.

routing

A frame containing one slot for each application that supports routing. The slot name is the symbol of the application.

userConfiguration

This frame contains information entered by the user in the Preferences application, such as name, phone number, etc.

Appendix E

NewtonScript Syntax

The definitions in this appendix are in a form of extended BNF and defined as follows:

`terminal`	Monospaced text indicates a word or character that must appear exactly as shown. Ambiguous terminal characters are enclosed in single quotes ('').
nonterminal	Italics indicate a word that is defined further.
[]	Brackets indicate that the enclosed item is optional.
{choose | one}	A group of words, separated by vertical bars (|) and grouped with curly brackets, indicates an either/or choice.

[]*	An asterik (*) indicates that the preceding item(s), which is enclosed in square brackets, can be repeated zero or more times.
[]+	A plus sign (+) indicates that the preceding item(s), which is enclosed in square brackets, can be repeated one or more times.

About the Grammar

The grammar is divided into two parts: the phrasal and lexical grammars.

In the phrasal grammar, whitespace is insignificant. `Space`, `tab`, `return`, and `linefeed` characters are considered whitespace. Comments are effectively considered whitespace. Comments consist of the characters between `/*` and `*/` (not nested), and between `//` and a `return` or `linefeed` character.

In the lexical grammar, the nonterminals are characters rather than tokens and whitespace is significant.

Because almost every construct of the language is an expression, many productions ending in expression are ambiguous; the ambiguity is resolved in favor of extending the expression as long as possible. For example, `while true do 2+2` is parsed as `while true do (2+2)` rather than `(while true do 2)+2`. The specific productions affected by this rule are function-constructor, assignment, iteration, if-expression, break-expression, try-expression, initialization-clause, return-expression, and global-function-decl.

Phrasal Grammar

input:
[*constituent* [; *constituent*]* [;]]

constituent:
{ *expression* | *global-declaration* }

expression:
{ *simple-expression* | *compound-expression* | *literal* | *constructor* | *lvalue* | *assignment* | *exists-expression* | *function-call* | *message-send* | *if-expression* | *iteration* | *break-expression* | *try-expression* | *local-declaration* | *constant-declaration* | *return-expression* }

simple-expression:
{ *expression binary-operator expression* | *unary-operator expression* | **(** expression **)** | **self** }

binary-operator:
{ *arithmetic-operator* | *relational-operator* | *boolean-operator* | *string-operator* }

arithmetic-operator:
{ **+** | **-** | ***** | **/** | **div** | **mod** | **<<** | **>>** }

relational-operator:
{ **=** | **<>** | **<** | **>** | **<=** | **>=** }

boolean-operator:
{ **and** | **or** }

string-operator:
{ **&** | **&&** }

unary-operator:
{ **-** | **not** }

compound-expression:
begin *expression-sequence* **end**

expression-sequence:
[*expression* [**;** *expression*]* [**;**]]

literal:
{ *simple-literal* | **'** *object* }

simple-literal:
{ *string* | *integer* | *real* | *character* | **true** | **nil** }

object:
{ *simple-literal* | *path-expression* | *array* | *frame* }

path-expression:
symbol [. *symbol*]*

Note: Each dot in ' *symbol* . *symbol* ... is ambiguous: it could be a continuation of the path expression or a slot accessor. NewtonScript uses the first interpretation: `'x.y.z` is one long path expression and not the expression: `('x).y.z`.

---

array:
'[' [*symbol* :] [*object* [, *object*]* [,]] ']'

frame:
'{' [*frame-slot* [, *frame-slot*]* [,]] '}'

frame-slot:
symbol : *object*

constructor:
{ *array-constructor* | *frame-constructor* | *function-constructor* }

array-constructor:
'[' [*symbol* :] [*expression* [, *expression*]* [,]] ']'

Note: '[' *symbol* : *symbol* (... is ambiguous: the first symbol could be a class for the array, or a variable to be used as the receiver for a message send. NewtonScript uses the first interpretation.

---

frame-constructor:
'{' [*frame-constructor-slot* [, *frame-constructor-slot*]* [,]] '}'

frame-constructor-slot:
symbol : *expression*

function-constructor:
  func ([*formal-argument-list*]) *expression*

formal-argument-list:
  symbol [, *symbol*]*

lvalue:
  { *symbol* | *frame-accessor* | *array-accessor* }

frame-accessor:
  expression . { *symbol* | (*expression*) }

array-accessor:
  expression '[' *expression* ']'

assignment:
  lvalue := *expression*

exists-expression:
  { *symbol* | *frame-accessor* | [*expression*] : *symbol* } **exists**

function-call:
  { *symbol* ([*actual-argument-list*]) |
  call *expression* **with** ([*actual-argument-list*]) }

actual-argument-list:
  expression [, *expression*]*

message-send:
  [{ *expression* | **inherited** }] { : | :? } *symbol* ([*actual-argument-list*])

if-expression:
  if *expression* **then** *expression* [;] [**else** *expression*]

Note: An `else` clause is associated with the most recent unmatched `then` clause.

---

iteration:
{ *infinite-loop* | *for-loop* | *foreach-loop* | *while-loop* | *repeat-loop* }

infinite-loop:
loop *expression*

for-loop:
for *symbol* **:=** *expression* **to** *expression* [**by** *expression*] **do** *expression*

foreach-loop:
foreach *symbol* [**,** *symbol*] [**deeply**] **in** *expression* { **do** | **collect** } *expression*

while-loop:
while *expression* **do** *expression*

repeat-loop:
repeat *expression-sequence* **until** *expression*

break-expression:
break [*expression*]

try-expression:
try *expression-sequence* [**onexception** *symbol* **do** *expression* [**;**]]+

local-declaration:
local *initalization-clause* [**,** *initalization-clause*]*

initialization-clause:
symbol [**:=** *expression*]

constant-declaration:
constant *constant-init-clause* [**,** *constant-init-clause*]*

constant-init-clause:
symbol **:=** *expression*

return-expression:
return [*expression*]

global-declaration:
{ *global initialization-clause* | *global-function-decl* }

global-function-decl:
{ **global** | **func** } *symbol* **(** [*formal-argument-list*] **)** *expression*

Lexical Grammar

string:
" *character-sequence* "

character-sequence:
[{ *string-character* | *escape-sequence* }]* [*truncated-escape*]

string-character:
<**tab** or any ASCII character with code 32–127 except '**"**' or '****'>

escape-sequence:
{ **** { **"** | **** | **n** | **t** } | **\ u** [*hex-digit hex-digit hex-digit hex-digit*]* **\ u** }

truncated-escape:
\ u [*hex-digit hex-digit hex-digit hex-digit*]*

symbol
{ { *alpha* | **_** } [{ *alpha* | *digit* | **_** }]* | '**|**' [{ *symbol-character* | **** { '**|**' | **** }]* '**|**' }

Note: Reserved words are excluded from the nonterminal symbol.

---

symbol-character:
<any ASCII character with code 32–127 except '**|**' or '****'>

integer:
[**-**] { [*digit*]+ | **0x** [*hex-digit*]+ }

real:
[-] [*digit*]$^+$. [*digit*]* [{ e | E } [-] [*digit*]$^+$]

character:
$ { *non-escape-character* |
\ { \ | n | t | *hex-digit hex-digit* |u *hex-digit hex-digit hex-digit hex-digit* } }

non-escape-character:
<any ASCII character with code 32–127 except '\'>

alpha:
<A–Z and a–z>

digit:
{ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }

hex-digit:
{ *digit* | a | b | c | d | e | f | A | B | C | D | E | F }

Operator Precedence

The precedence of operators, from highest to lowest, is shown in Table 5.1, "NewtonScript operators grouped in precedence order.," on page 137.

Appendix F

Application Issues

Setting Application Bounds Based on the Screen Size

If you wish to make the size of your application base view dependent on the size of the screen, you will need to add a `viewSetupFormScript` to your base view. You should set some reasonable limits on the maximum size of your application as well; the screen size on some future Newtons may be arbitrarily larger than you really want.

Here is an example `viewSetupFormScript` that incorporates both a particular Newton's screen size and a maximum size:

```
func()
begin
   constant kPixelsPerInch := 80;
   local maxWidth := 10 * 80; // 10 inches
   local maxHeight := 11 * 80; // 11 inches

   local params := GetAppParams();
   self.viewBounds := RelBounds(
      params.appAreaLeft,
      params.appAreaTop,
      Min(maxWidth, params.appAreaWidth),
      Min(maxHeight, params.appAreaHeight)
   );
end
```

The `GetAppParams` function returns a frame with five slots:

`appAreaLeft`	The X coordinate of the top left screen corner.
`appAreaTop`	The Y coordinate of the top left screen corner.
`appAreaWidth`	The full width of the screen in numbers of pixels.
`appAreaHeight`	The full height of the screen in numbers of pixels.
`buttonBarPosition`	This is a symbol with one of four values (`'left`, `'top`, `'right`, `'bottom`). The symbol indicates the position of the Newton's permanent button bar. You can use the information to locate your application relative to the bar.

Creating Unique Application Symbols and Names

There are a number of names and symbols (like application and soup names) that need to be unique among applications. There is a mechanism for avoiding conflict: each developer should register a signature (usually a company name) with Apple Computer, Inc. The symbols or strings that must be unique end with a colon (:) and then a particular unique portion of the signature. Thus guaranteeing signature uniqueness, each developer can generate unique symbols and names for

every product developed (for example, "ProductA:Signature" and "ProductB:Signature").

Here are some examples that utilize the registered signature "Calliope". These are the symbols and names that must be unique to a particular application.

application symbol	Among other things, there is a slot in the root view with this symbol value that points at the base view of the application. For example, `'|WaiterHelper:Calliope|`.
package name	This is shown in the Remove Software picker. Be careful—your users see this name. For example, `"WaiterHelper:Calliope"`.
soup name	You may not realize it, but your user sees this name in the Edit Folders slip and in Newton Connection. For example, `"WaiterHelper:Calliope"`.
System soup tag	Any preferences data an application stores in a soup is stored in the "System" soup with a tag slot containing a unique string. The user won't normally see this. For example, `"WaiterHelper:Calliope"`.
Extra slots in existing soups	When an application wants to add a slot to entries in the "Names" soup, for instance, it would create a frame with its application symbol. You then add data to that frame. For example, `'|WaiterHelper:Calliope|`.
Routing formats	Applications that support printing or faxing add a frame to the root view. For example, `'|WaiterHelper:Calliope|`.

To register your unique signature with Apple Computer, Inc., you will need the following information:

- Name
- Contact person
- Mailing address

- Phone
- Email address
- Desired signature, first choice
- Desired signature, second choice

Send the information to:
AppleLink: PIESYSOP
Internet: PIESYSOP@applelink.apple.com
Mail:
Apple Developer Support
20525 Mariani Avenue
Cupertino, CA 95014

Appendix G

Using Newton Toolkit

Newton Toolkit:
Development Environment of Champions

—*Newton Toolkit*

This appendix is about Newton Toolkit (hereafter NTK), the development environment for Newton. NTK manages the entire life cycle of an application: you create projects, layout templates, build your application, download to the Newton, and debug all within NTK.

Installing NTK

Three files need to be installed in your system folder for NTK to work properly:

Apple Modem Tool	This Comm Toolbox communications tool is used by NTK to communicate with your Newton using a serial port. Install it in the Extensions folder.
AppleTalk ADSP Tool	This Comm Toolbox communications tool is used by NTK to communicate with your Newton using LocalTalk. Install it in the Extensions folder.
Newton Toolkit Font	This font is used to preview within NTK how things will look on the Newton. Install it in the Fonts folder (if using System 7.1 or later), or in the System file (if using System 7.0 or earlier).

These are the files in your Newton Toolkit Folder:

Newton Toolkit	The Development Environment of Champions.
GlobalData	A text file to which you can add constant definitions that are then available to all your projects.
EditorCommands	File defining commands used for editing text within NTK.
NTK Definitions	A text file containing a huge number of constants that you can use in your code.
Toolkit App	A package for the Newton that allows you to connect the Inspector and download NTK applications.
Platforms Folder	A folder for containing files for each of the Newton platforms.
MessagePad	The platform file for the Newton Message Pad, first Newton in a family of products.

Launching NTK

The first time you launch NTK you'll see the dialog shown in Figure G.1. If you already have an existing project, you can select it from this dialog. New projects are handled differently—for now, simply click the Cancel button.

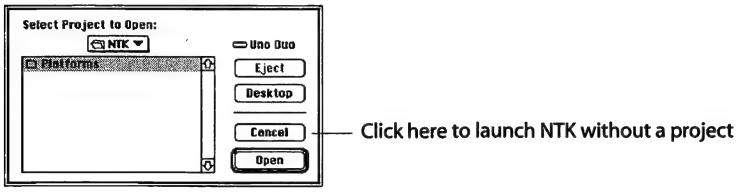


Figure G.1 Initial NTK dialog.

Setting Your Connection Type

The Newton does not need to be connected to the Macintosh while you write and build your applications. But if you want to download or debug the application, you need to connect the two machines. You can connect the Macintosh to the Newton in one of two ways: through a serial connection or using LocalTalk. If you have a choice, the serial connection is preferable. You specify your connection choice using the Toolkit Preferences dialog, which is found in NTK's Edit menu (see Figure G.2).

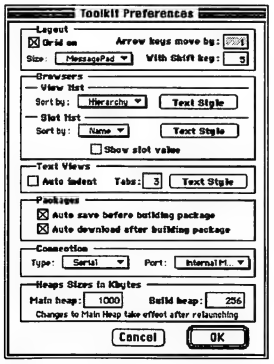


Figure G.2 Toolkit Preferences dialog.

Choosing a Serial Connection

If you are using a serial connection, you'll need to specify which serial port you are using. For most Macintoshes, the available choices are the printer or modem port (see Figure G.3).

Note: If you are using the built-in serial port of a PowerBook Duo, select the Printer-Modem port. If you have a PowerBook internal modem, make sure the modem is set to normal (not compatible). In addition, make sure Appletalk is off.

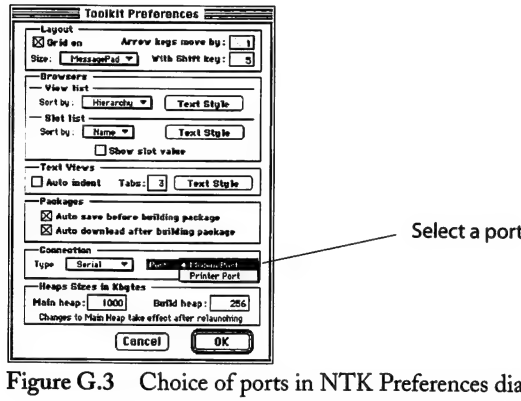


Figure G.3 Choice of ports in NTK Preferences dialog.

Hook up a serial cable from your Newton to the port you specified on your Macintosh (use a Macintosh/Imagewriter II cable, available from an Apple dealer as model M0197—part number 590-0552-A). Then, choose Install Toolkit App from the Project menu in NTK. On the Newton, open the Extras drawer and tap the Connection icon (see Figure G.3).

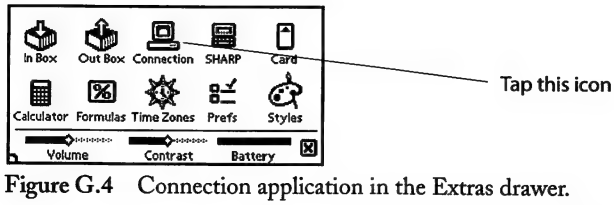


Figure G.4 Connection application in the Extras drawer.

After choosing the Macintosh serial radio button, tap Connect (see Figure G.3). In a few seconds, the NTK Toolkit App should appear in the Extras drawer.

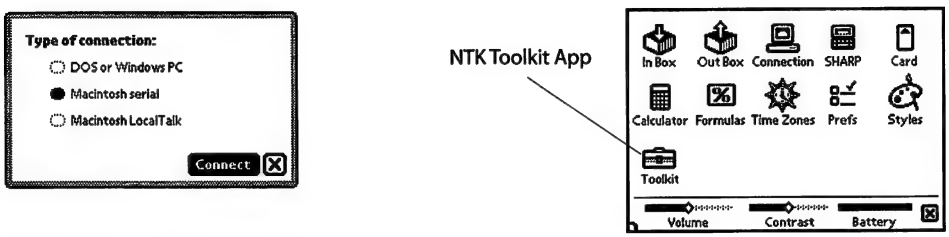


Figure G.5 Connection slip.

NTK Menus

Here is a brief look at the menu options available in NTK. Many of them are standard to the Macintosh and require no additional information. This section is just an overview of the NTK-specific items so that you can get the flavor of the whole environment. Specific items are covered later on in this appendix.

The File Menu

In the File menu, you select what type of layout file you wish to create (see Figure G.6). You will most commonly choose New Layout, which is where you draw out your application's view templates. You select New Proto Template when you want to make a user proto.

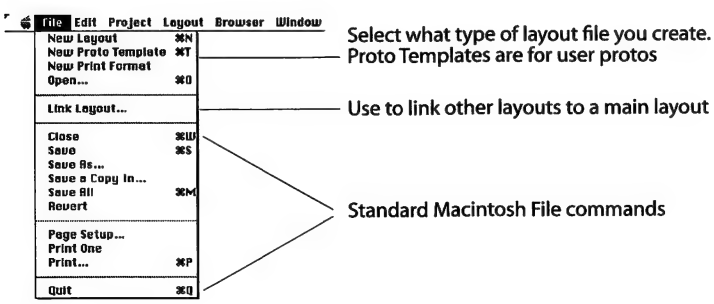


Figure G.6 NTK's File menu.

The Edit Menu

The Edit menu contains standard copying and editing commands as well as some selection and find tools (see Figure G.7). The Toolkit Preferences dialog is also accessed from this menu. As shown in Figure G.2, Preferences is where you specify connection settings and other project settings like auto saving and layout screen size.

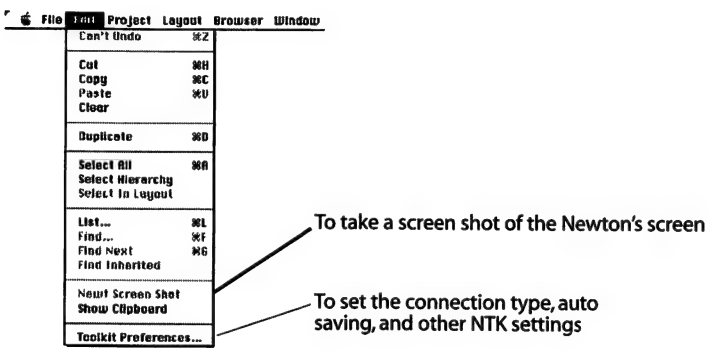


Figure G.7 NTK's Edit menu.

The Project Menu

The Project menu contains NTK's project management tools (see Figure G.8). You need to have the project window frontmost to be able to access many of these tools, and you must add the files to your project before you can either build or download the application.

Further, if a file is not added to a project it will not be part of a build. You can tell which files are part of a project by the list in the project window (see Figure G.16).

Four important items in this menu include Mark as Main Layout, Project Data, Settings, and Export Package to Text. You should only have one file marked as a main layout, and this should be your main application layout. The Project Data is the file where you keep various settings like symbol and constant definitions and scripts that get run prior to installation and removal of your application. In the Settings dialog, you specify various aspects of the application, like the name, symbol, and icon. Export Package to Text creates a text version of your project (this conversion is one-way; NTK cannot read text versions of a project).

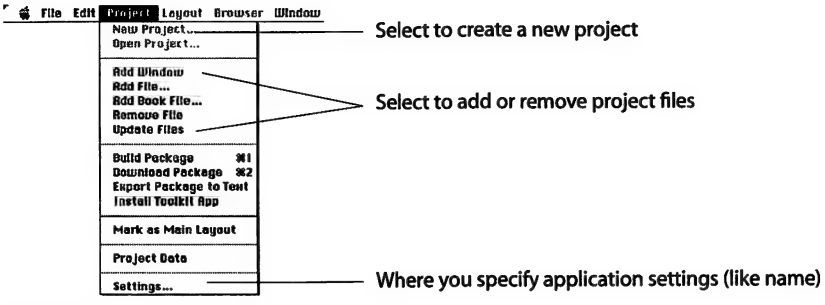


Figure G.8 NTK's Project menu.

The Layout Menu

The Layout menu contains items that control layout and template graphical arrangement (see Figure G.9). Preview is a particularly nice feature. It gives you a rough idea of what your application views will look like on the Newton without actually having to download. ⌘-Y toggles Preview on and off.

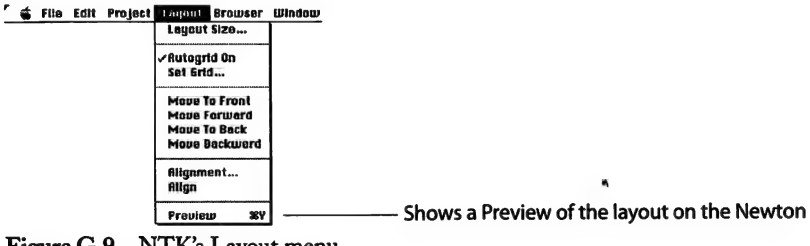


Figure G.9 NTK's Layout menu.

The Browser Menu

Items in the Browser menu consist of a slot creation dialog and various display options for templates and slots (see Figure G.10). The Template Info dialog is where you name and declare templates.

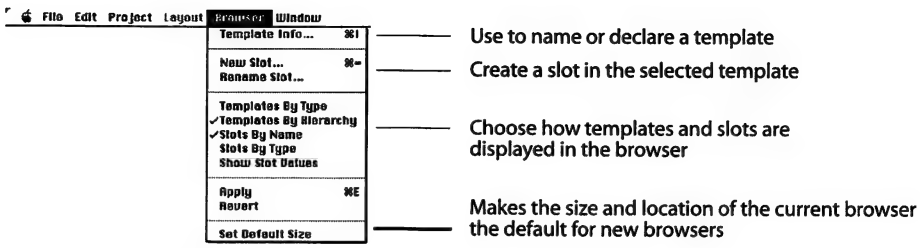


Figure G.10 NTK's Browser menu.

The Window Menu

Within the Window menu, the most important items are connecting to the Inspector and creating a New Browser (see Figure G.11). You select Connect Inspector (not Open Inspector) to establish your connection to a tethered Newton. Once you have created a layout window with drawn-out templates, you access the templates and their slots in Browsers.

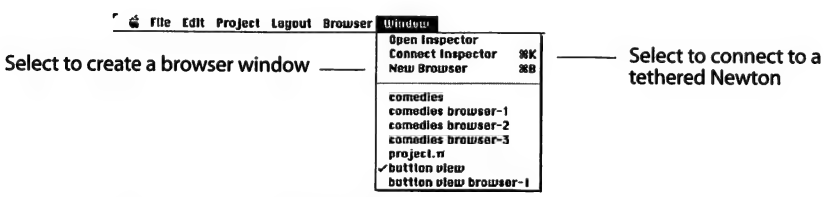


Figure G.11 NTK's Window menu.

Creating a Project

To create a project, use New Project from the Project menu (see Figure G.3). You are prompted with a dialog in which you must specify a name and folder for the project (see Figure G.3).

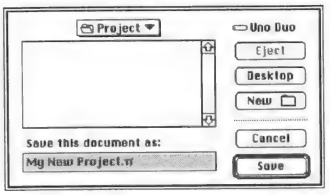


Figure G.12 New Project dialog.

As there are no files in the project, the project window is empty (see Figure G.3).

Figure G.13 Empty project window.

NTK Project Files

There are five different types of files that you might have in your NTK projects: Layout files, Proto files, Resource files, Book files, and Print Format files (see Table G.1). Later in this appendix, we will discuss the two types of project files that you create within NTK—Layout and Proto files.

File type	*Contents*
Layout files	Contain a collection of views.
Resource files	Contain sounds or pictures.
Proto files	Contain a collection of views which can be used as a basis for other views.
Book files	A Newton Book Maker file that is added to the project. NTK builds a Newton Book using this file.
Print format files	Used for supporting printing and faxing.

Table G.1 Files in an NTK project.

Adding Files to a Project

You can add files to the project in one of two ways: using Add File or Add Window.

Add File

Add File is the most general-purpose way to add a file to a project and is what you usually use. To use Add File to add a layout window to a project, do the following:

1. Save your layout by choosing Save from the File menu. Name the file.
2. Save the layout in the same folder as your project (see Figure G.14).
3. Choose Add File from the Project menu.
4. Select the file you just saved in the Add File dialog (see Figure G.15).

Your file is now part of the project, as you can see in the project window in Figure G.16.

Note: By convention, layout files end with ".t". One way to remember this is that the word layout ends in "t".

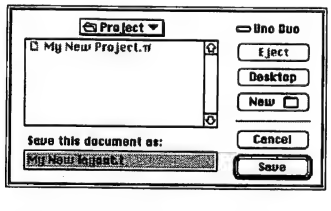


Figure G.14 Saving a layout.

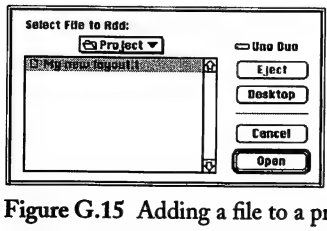


Figure G.15 Adding a file to a project.

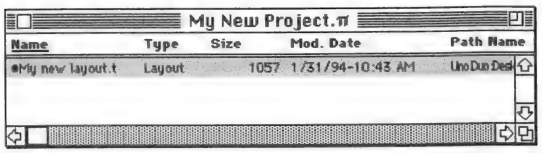


Figure G.16 Project window with a new file.

Add Window

Add Window is a shortcut for adding a new layout window to the project. To use it, simply make sure that the new layout window is the frontmost window on your screen. Then select Add Window from the Project menu. Add Window is dimmed if the frontmost window is not a layout or if it is already part of the project.

The Add Window command will prompt you with a Save dialog if the file has never been saved.

Creating a Layout

You create the templates that correspond to Newton views in NTK layout windows.

1. To create a layout window, select New Layout from the File menu.

   NTK then displays an empty layout window, along with a Tool Palette (see Figure G.17). You create templates inside this layout by graphically dragging them out on the layout screen.

2. To begin, select protoApp in the palette by clicking on its icon or selecting it from the popup menu (see Figure G.17).

3. Once the protoApp is selected, go to the window and drag out a rectangular shape (see Figure G.18).

   This template shape will correspond to the view displayed on the Newton; it will have the same dimensions and screen location.

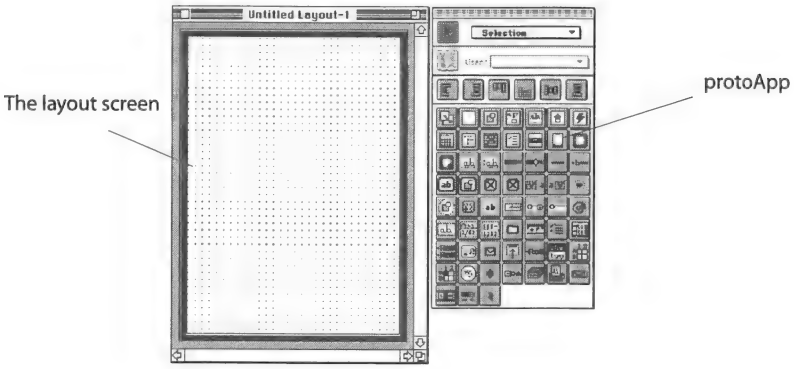


Figure G.17 Empty layout window with tool palette.

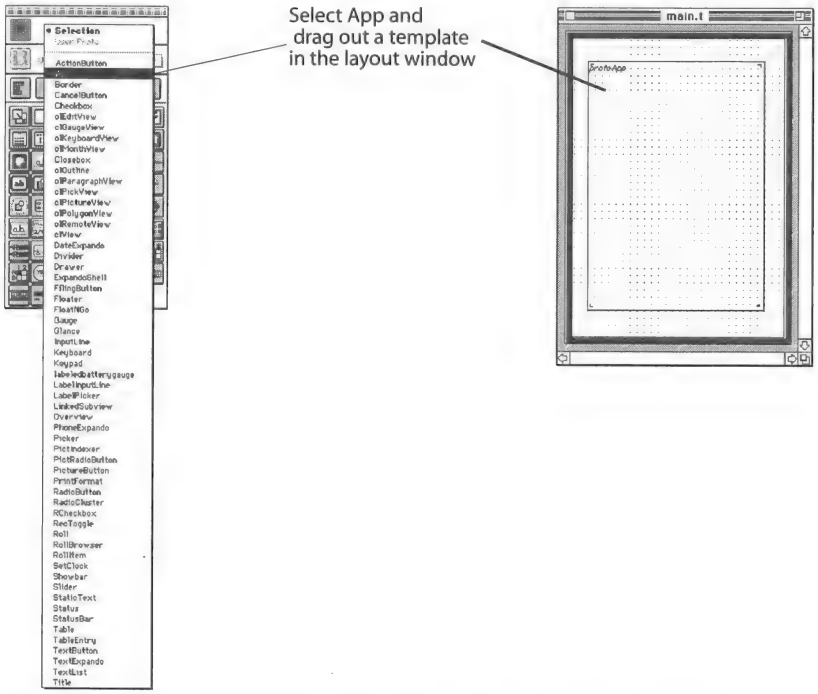


Figure G.18 Selecting protoApp in the tool palette and creating a template.

All of the views that you have in your application are usually created in NTK as templates in these layout windows. For each template in your application, you go through the same twofold process:

- First, you select a template type.
- Second, you drag out its shape in the layout window.

Linking Layouts

When you have too many templates to fit comfortably in one layout, or you have templates that will be visible in the same screen location, it is usually easier to create them in different layout files and then link the layouts to each other. As a rule of thumb, you will have a main layout with your other layouts are linked to it. To link layouts, follow these steps:

1. Make sure that the main layout file and any sub-layout files are added to the project (see Figure G.19).
2. While in the main view, select the Linked Subview tool in the Tool palette (see Figure G.19).
3. Drag out a small shape in the main layout (see Figure G.20). The size of this linking template does not affect the view size—location and size are determined by the templates in the other layout file.
4. Select this new template and select Link Layout from the File menu (see Figure G.20).
5. In the dialog, select the name of the layout file you wish to link (see Figure G.20).

The linked subview template in the main layout should now be linked to the layout (see Figure G.21).

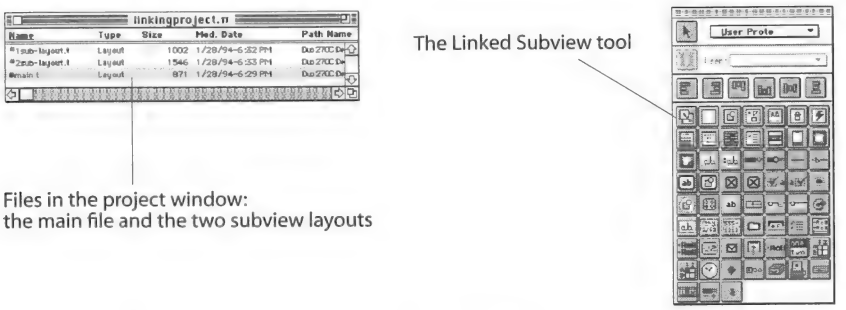


Figure G.19 The project window and the Tool Palette.

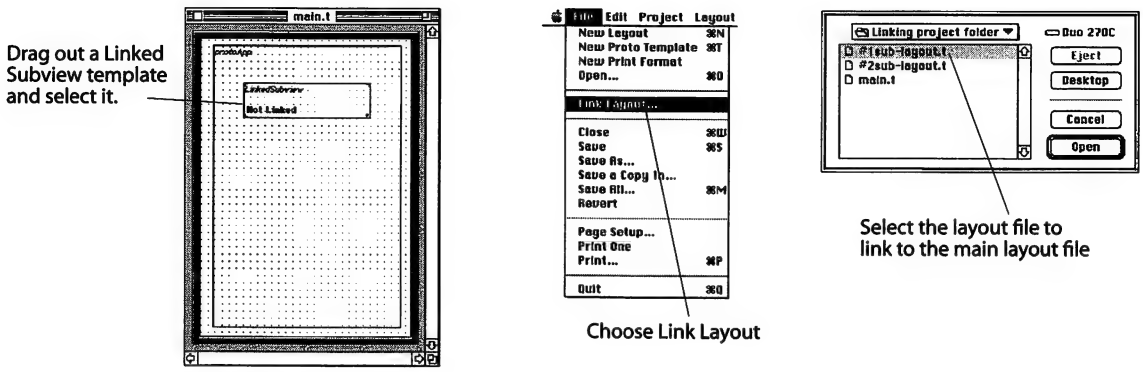


Figure G.20 Linking "#1sub-layout.t" to the `protoApp` template in "main.t".

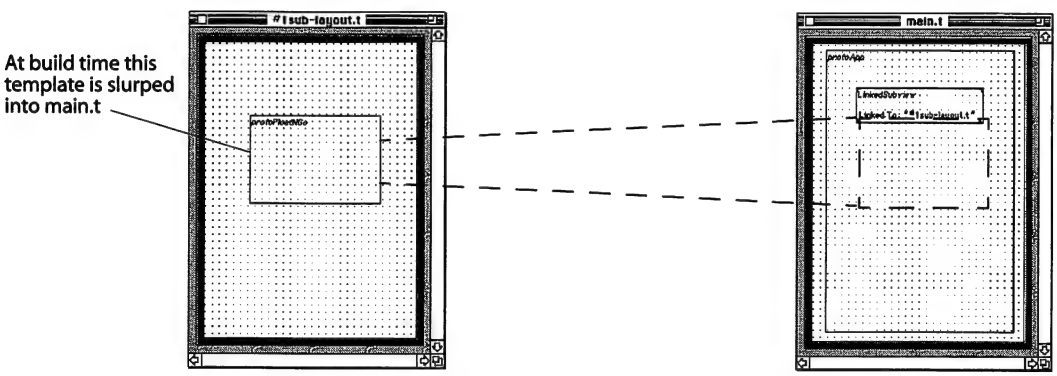


Figure G.21 Main layout, "main.t", and linked layout, "#1sub-layout.t".

NTK and Linked Subviews

Linking was added to NTK as a convenience for Newton programmers and to aid in application design. This should be fairly comprehensible when you understand what NTK does with your layouts. At build time, NTK reads all the templates for the main layout, and for any linked layouts, reads those layout files (recursively continuing to read linked layouts). It creates a large tree of templates, which it sends to the Newton (see Figure G.21).

Creating a User Proto

As you learned in the discussion of protos in Chapter 4, the way to reuse templates and build code libraries of application components is to use protos. Proto creation is remarkably similar to regular layout creation; the only difference is the type of layout file you select in the File menu.

To create a proto, select New Proto Template from the File menu and you will get the different-looking layout screen you see in Figure G.22.

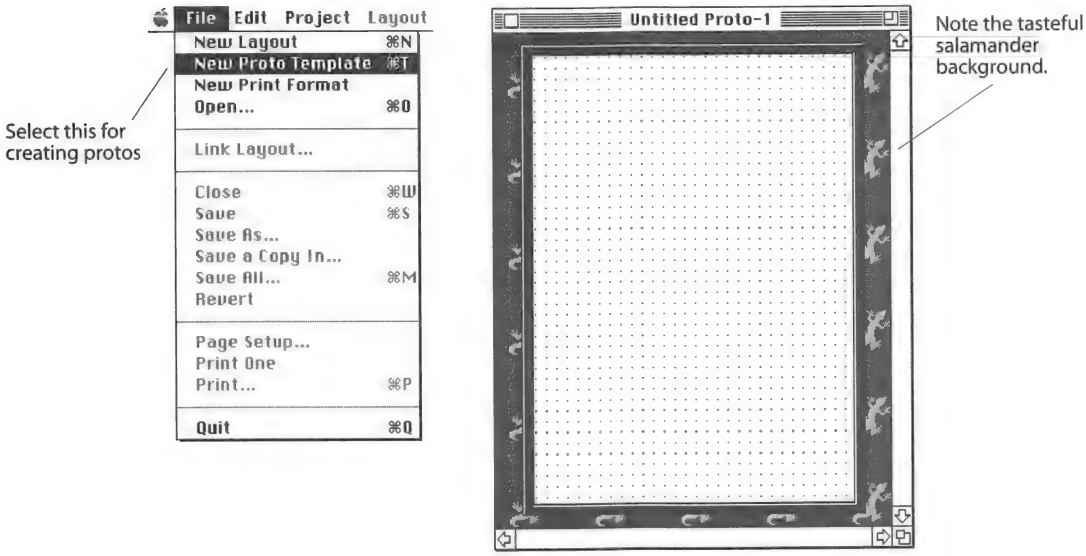


Figure G.22 Creating a new proto template.

You have the same tool palette available to you in creating your proto. Further, the actual dragging out of templates in the proto is identical to template creation in a regular layout window. Once you have completed your proto you must add it to your project using Add Window or Add File from the Project menu. Once the new proto is added to the project, it is available for use in the tool palette with the other tools (see Figure G.23). User protos are displayed and picked from their own list in the tool palette (see Figure G.23), however.

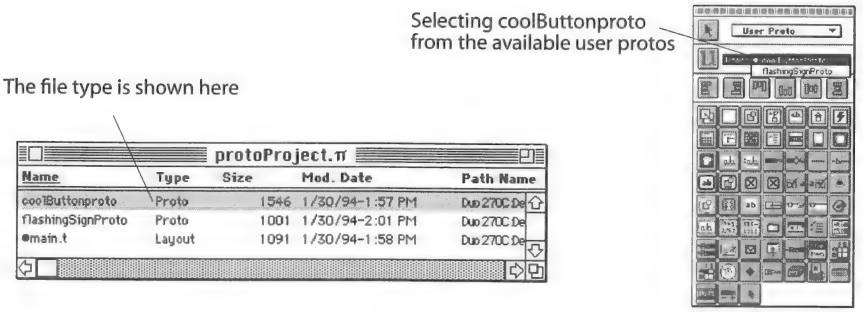


Figure G.23 A project and tool palette with User protos.

Note: Proto file names should contain no special characters or spaces. Unlike layout files, you might use the name of the proto in your code. If you have special characters in the name, then you must reference the proto with vertical bars. It is easier to avoid special characters and spaces in the first place.

Adding a User Proto to a Layout

To add a user proto to a layout is very simple:

1. Select the user proto in the list of user protos (for example coolButtonproto).

2. Select the user proto icon and drag out a template in the layout window (see Figure G.24). You must reselect the user proto icon each time you want to drag out another proto template.

Notice that it does not matter what size you attempt to make the proto template. It will draw out with the screen location and size specified in the original proto (see Figure G.24). The proto template is created with only one slot, the `_proto` slot. If you want to change the size of the template, add a `viewBounds` slot to the newly created template.

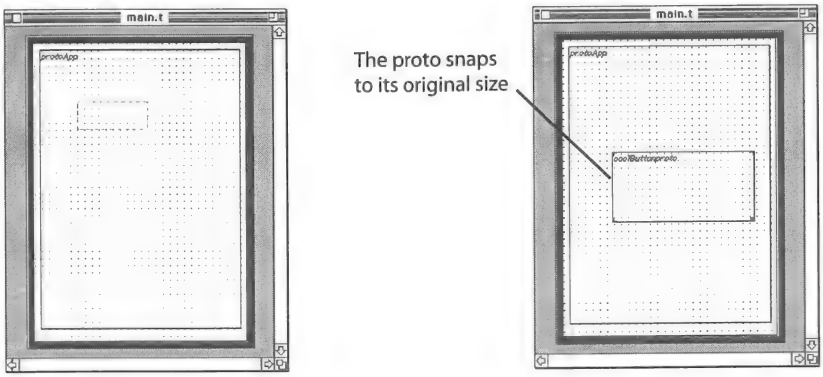


Figure G.24 Dragging out a coolButtonproto template.

Note: If you have no `viewBounds` slot in your proto, then NTK will create your template with both a `_proto` slot and a `viewBounds` slot.

Creating and Modifying Templates

Once you have your layout window created you need to populate it with some templates. As we said before, template creation is a two-step process that you repeat for each template you want to create:

1. Select a user proto, view class, or proto from the tool palette that corresponds to the type of template you want to create.
2. Drag out a shape in the layout window that you want the template to have. Move the template to the correct screen location if it is not already in the right place.

Putting Templates in a Parent–Child Hierarchy

The first template that you create in a layout window serves as the main parent template. There can only be one parent of this type in a layout. If you try and cre-

ate another template that isn't a child of the first template, you will get the NTK alert shown in Figure G.25.

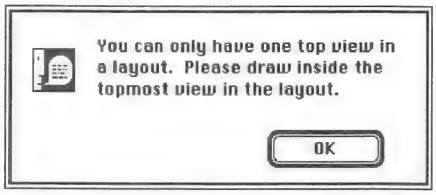


Figure G.25 NTK alert, warning of a parent clash.

Just remember that every layout file must have a base parent template—NTK insists on the single-head-of-household filing status. Inside of that parent, however, you can have as many children as you wish.

To make sure that a template is created as a child of another template, you simply drag out the child template shape inside the parent as in Figure G.26. You can also change the template hierarchy if you don't get it positioned correctly on the first try (see "The Template Area of the Browser" on page 369).

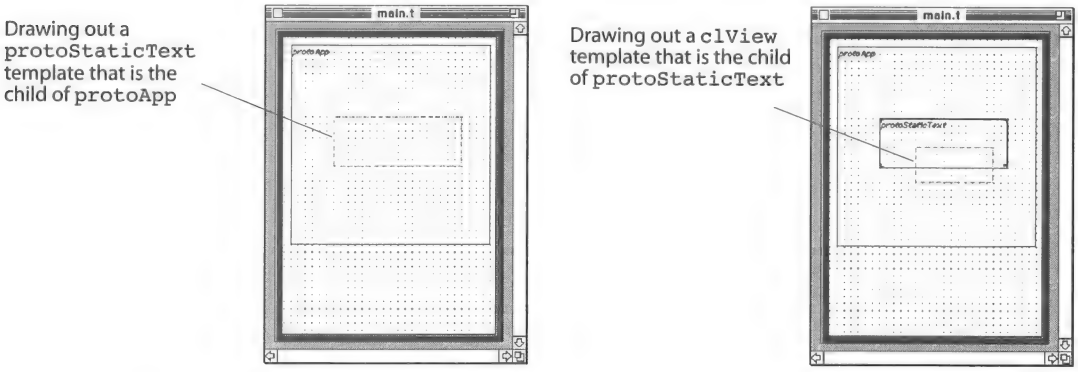


Figure G.26 Dragging out child templates.

Notice in Figure G.26 that the child template of the `protoStaticText` template is drawn longer than its parent. This does not change its place in the hierarchy, which is set by where you first start to draw a template. If a child view is longer or wider than its parent, it will draw outside its parent on the Newton,

unless the parent had its `vClipping` bit set in `viewFlags`. In the latter case, the child view will be clipped off at the edge of the parent. NTK always displays children clipped to their parent, regardless of the `vClipping` bit setting.

Moving and Changing Template Size Graphically

To move a template, simply click on it, and its four corner edit squares will become active. If you want to move the template, click, hold the mouse down, and drag the template to the new position. You can tell that you are moving and not resizing the template because the cursor icon turns into a little moving hand.

If you want to resize the template, drag the bottom right edit square (see Figure G.27) to be smaller or larger. You can tell that you are resizing and not moving because the cursor turns into a diagonal line with arrow heads.

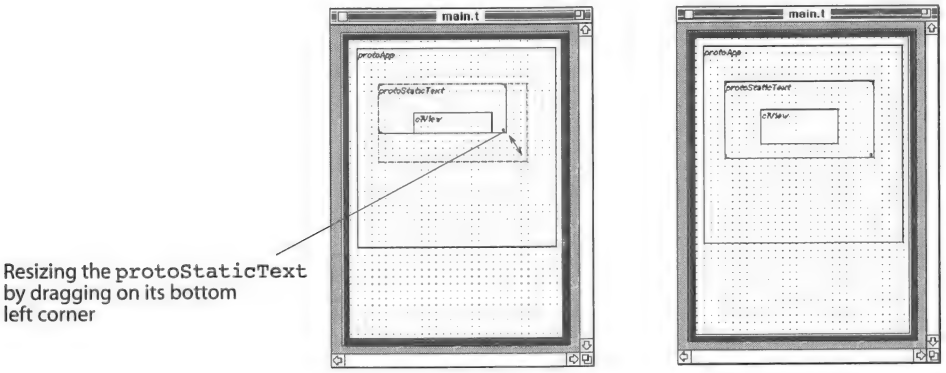


Figure G.27 Resizing the `protoStaticText` template.

Naming Templates

To name a template, select it in the layout window and choose Template Info from the Browser menu. Type the name of the template in the dialog (see Figure G.28). You can see the name of the template in the layout window if the template is large enough.

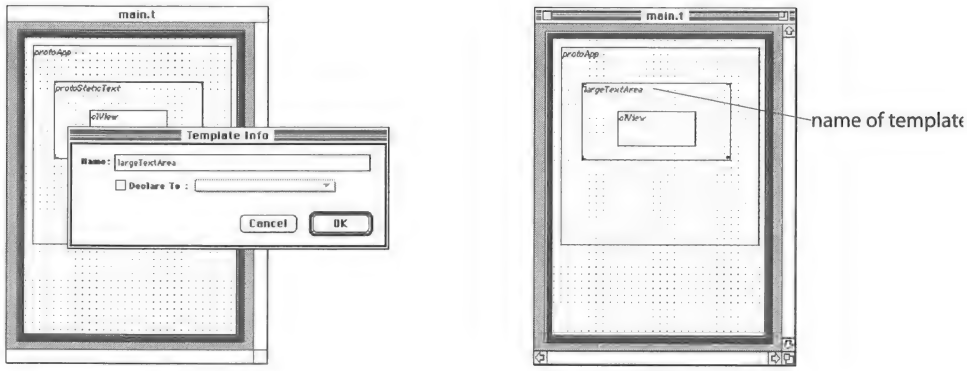


Figure G.28 Naming a template.

Using Declare To on a Template

You can also declare templates in the Template Info dialog. You declare a child template to a parent or other ancestor template, if it needs access to the child. Here is how you declare a template:

1. Make sure the parent is named. Select the template and name it in Template Info (see Figure G.29).
2. Open Template Info on the child template, name it, and check Declare To. Select the parent's name in the list (see Figure G.29).

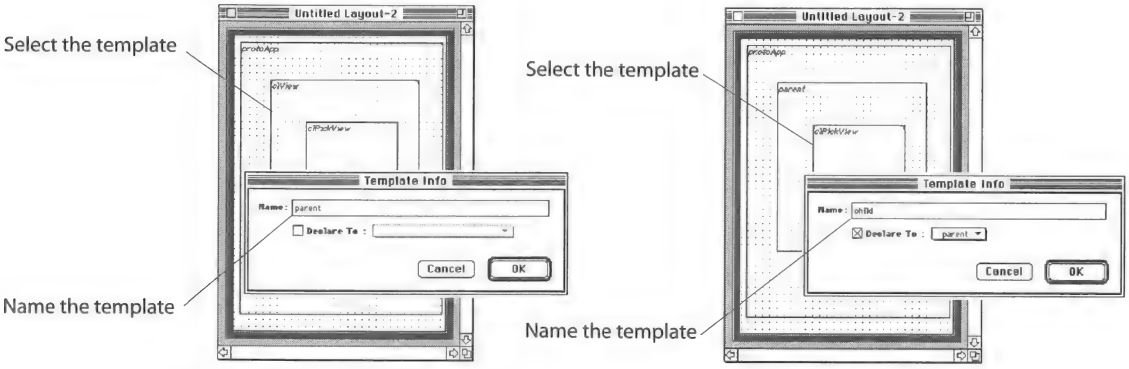


Figure G.29 Declaring a child to a parent.

If the child template is large enough, it will indicate its declared status in the layout window as well (see Figure G.30). You can declare a child to any ancestor template in its layout—not just its immediate parent. *If you forget to name the child, the Declare To checkbox remains dim. If you forget to name the parent, its name won't appear in the popup.*

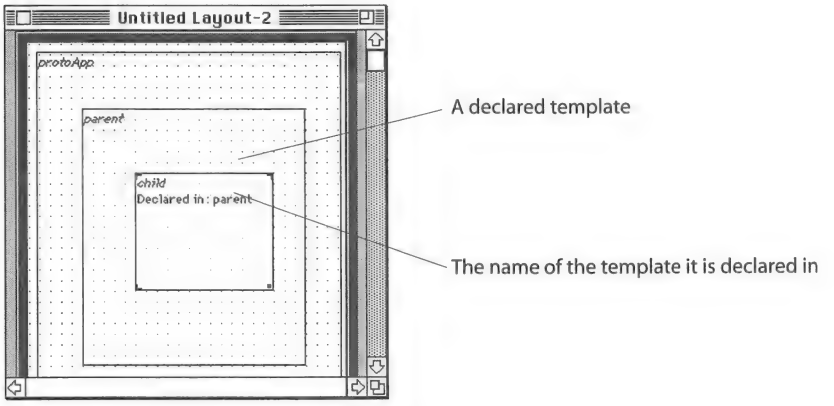


Figure G.30 A child template declared to its parent.

Removing Templates

If you want to get rid of a template, just select it in the layout window and select Cut from the Edit menu. The template is now gone. In cases where it is hard to select a template in the layout window, you can also do the following:

1. In the browser window, double-click on the template you want to remove.
2. Make the layout window the frontmost window and then select Cut from the Edit menu. The template will be removed from the layout.

An Introduction to the Browser

You just encountered a new term in the last section—the Browser. This is the tool provided by NTK to allow editing the slots in a template. The browser window is created by selecting New Browser in the Window menu. The browser window has three parts (see Figure G.31). The left top shows a list of templates and their hierarchical relationship. The top right shows the slots in the template that is selected

on the left. And the bottom of the browser shows the slot editor for the selected slot. In Figure G.31, you see a browser window with a `protoApp` template selected on the left, its `title` slot selected on the right and the contents of its `title` slot shown below.

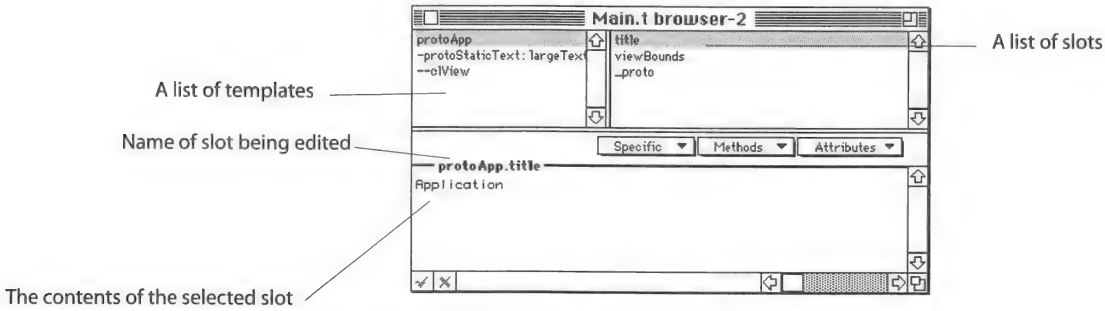


Figure G.31 The browser window.

New Browser window creates a browser for the selected templates in a layout. The topmost parent you have selected is the topmost template shown in the browser window (see Figure G.32).

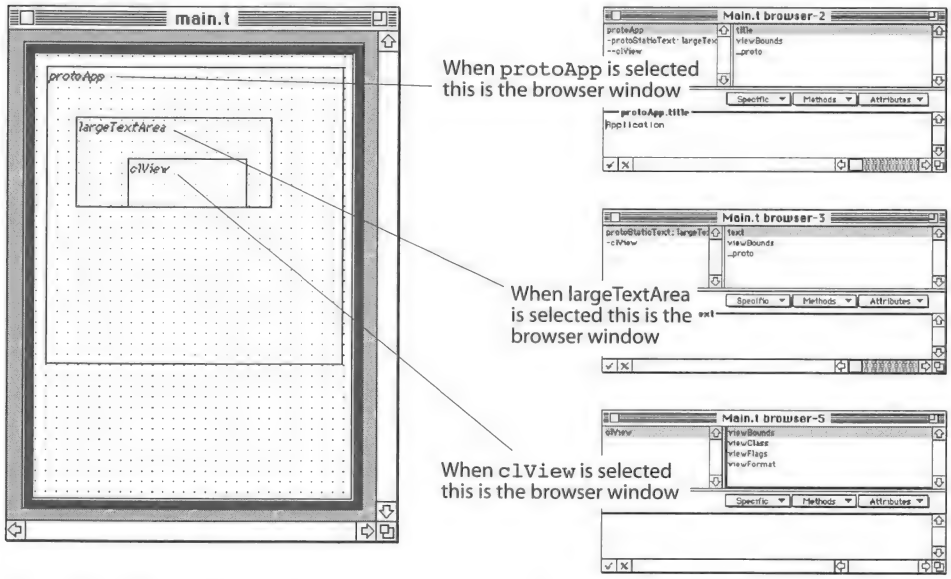


Figure G.32 Creating browsers on various templates in a layout.

Note: Another way to open a browser window is to option–double–click on a template in a layout window.

You will spend a majority of your time in the browser. It is here that you will add the methods to your application and any specialization in each template's appearance.

The Template Area of the Browser

In the template area of the browser you can see the hierarchical relationship between the templates and change them as well. You can select a template and then use a combination of the option and arrow keys to move the template around in the structure. The up and down arrow keys will move the template up or down in order among its siblings. The left arrow key moves a template out to the next level (its old parent's level) and the right arrow key moves a template into the next level (makes it a child of its previous sibling).

A template's sibling is the one immediately above it in the same level of the template list (see Figure G.33). If the template immediately above is its parent, the template has no previous sibling.

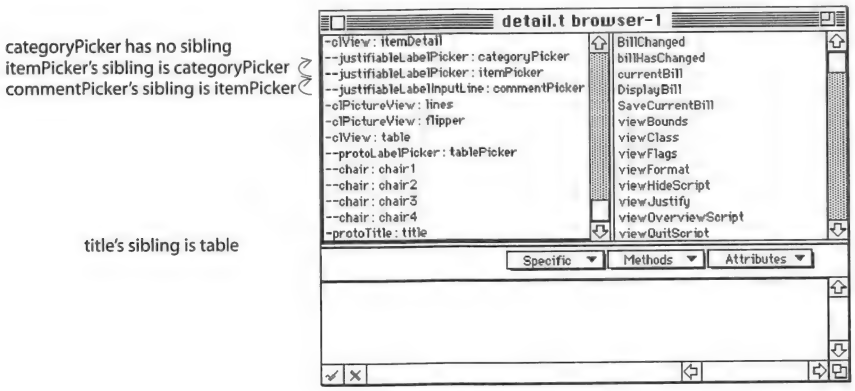


Figure G.33 Identifying a template's sibling.

Selecting Templates

Sometimes it can be difficult to select a template in the layout window (for example, when a child is as big as a parent template). If you need to select this type of pesky template, do the following:

1. Double-click on the template in the browser.
2. Make the layout window the frontmost window.

The Template Hierarchy

In the template list in the browser you can also see the parent–child relationship between templates. The dash symbols represent the level of ancestry:

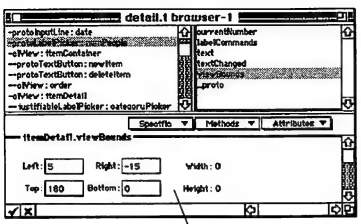
no dash	the main template displayed in the browser
-	child of the main template
--	grandchild of the main template, or child of the last - template

The Slot List of the Browser

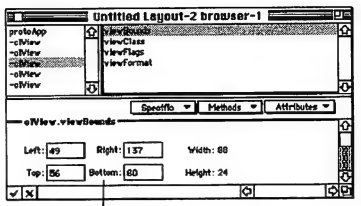
As we said, the right side of the browser displays the slots of the template selected on the left. You can select the slot, edit it, remove it, or paste another slot into that template when the top right of the browser is selected.

Editing an Existing Slot

To actually edit a slot from the slot list, *you need to double-click on the slot name*. You may find this a difficult aspect of NTK to work with at times. For instance, double-clicking can be annoying when you are resizing a number of templates—imagine you are in the middle of editing the third of 10`viewBounds` slots. It is difficult to remember which slot you are changing or whether it corresponds to the one selected in the list (see Figure G.34). This points out another reason why it is useful to name the templates you are working with. Without a name, you would have no way of knowing which of the five `clView` templates in Figure G.34 you are editing without double-clicking again in the slot editor. If templates are named, then the name of the template and its slot are always displayed in the slot editor.



This is itemDetail's `viewBounds`—not numPeople's

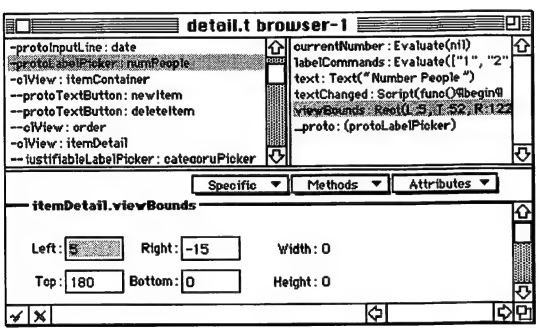


Q: Whose `viewBounds`?
A: It is actually that of the second `clView`.

Figure G.34 Slot editors don't always match the selected slots.

Displaying a Slot's Values

You can also display the values after the slots in the list by selecting Show Slot Values from the Browser menu. This gives you an abbreviated version of the slot contents, as displayed in Figure G.35.



Displaying slot values

Figure G.35 Displaying slot values in the slot list.

Deleting and Renaming Slots

You can delete a slot by clicking on it in the slot list and selecting Cut from the Edit menu.

You can rename a slot by clicking on it in the slot list and selecting Rename Slot from the Browser menu. Make sure not to use a name of a predefined slot unless you intend to override that slot.

Adding Slots

You can add new slots to a template in the slot editor. The most convenient method for adding a slot is only available for predefined slots. For these slots, you can select the slot by its name in one of the three popup menus: Specific, Methods, or Attributes. Figure G.36 displays the contents of these three popups.

Note: The Specific popup menu's contents will vary, depending on the type of template. The specific items are those slots that are standard optional additions to a template.

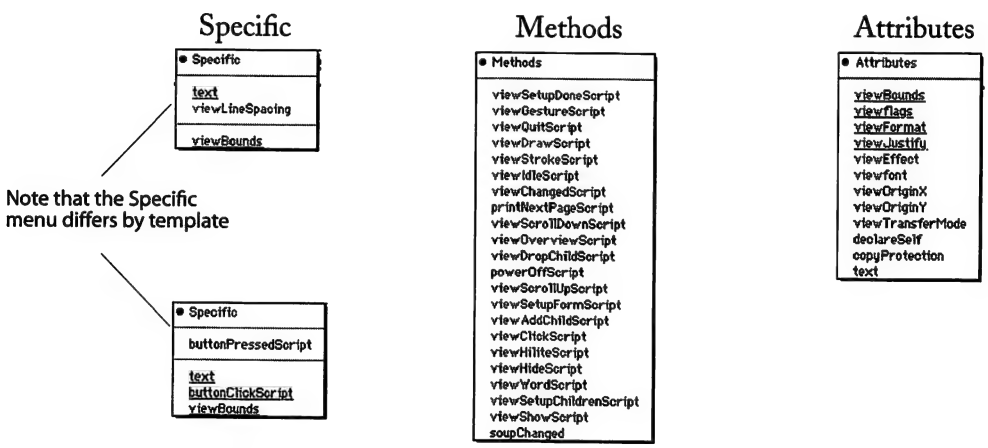


Figure G.36 Popup menus in the slot editor of the browser.

These slots are all the standard slots that NTK provides to you by name. This is roughly how the slots are grouped in the popup menus:

Specific — These are the slots specific to a particular type of template. They are slots in the proto that you might want to override in a particular template. For example, the `viewFont` slot in `protoStaticText` is available in the Specific menu.

Methods	These are standard methods associated with views.
Attributes	These are slots that are used with many different kinds of templates.

The New Slot Dialog

You can also add a slot using the New Slot dialog, which is found in the Browser menu. When you select New Slot in the menu, you get the dialog shown in Figure G.37. It is also in this dialog that you will add all of your own methods to a template. At the top of the dialog are the same popup menus available in the slot editor. Below is a text area to enter the name of your slot.

To the right of the name is another popup, where you can specify one of eight types of slot editors:

Evaluate	A generic editor, preset to `NIL`.
Custom	A custom editor (currently disabled).

Script	A editor for functions, preset to hold
	`func()`
	`begin`
	`end;`

Text	A editor for text, preset to surround the slot value with double quotes, " ".
Number	An editor for integers.
Boolean	An editor for `True` or `NIL` values.
Rectangle	An editor to enter left, right, top, and bottom values.
Picture	An editor with a file picker and a resource display area.
Font	A font editor (currently disabled).

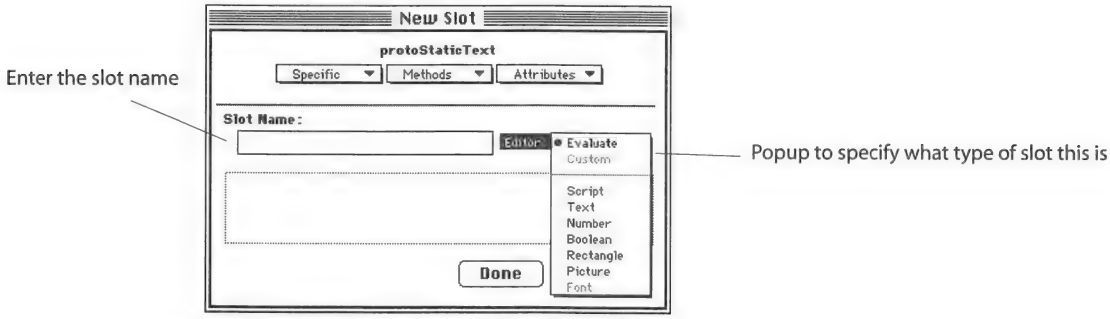


Figure G.37 NTK's new slot dialog.

The Slot Editor

The slot editor is located in the lower portion of the browser window (see Figure G.38). Its ingredients are fairly simple. The name of the slot being edited is displayed at the top left of the editor. The value found in the slot is displayed in the editor window. Below the editor window are two checkboxes, Apply and Revert.

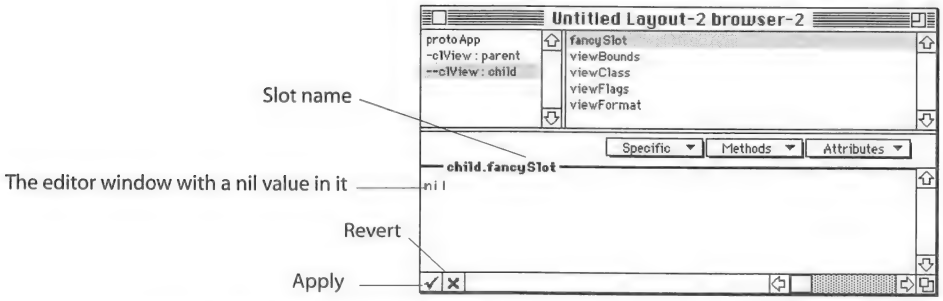


Figure G.38 Editing `fancyslot` with an Evaluate slot editor.

The Apply and Revert Buttons

Think of the editor window as an interim work area. Whatever you enter there is temporary until you take the further step of writing it to the slot—which you do by clicking the Apply button. You can go edit another slot or work in another part of the project; the slot you were working on will remain unchanged if you do not

hit the Apply button. If you have Auto Apply selected in the Settings dialog, then changes in the editor window are automatically applied when ever you build or otherwise leave that editor window.

Hitting the Revert button lets you change your mind about a proposed change—the contents of the slot editor window are removed and the slot's original value is redisplayed. You can no longer revert once you have clicked Apply; the Apply button takes a change you have entered in the editor and writes that to the slot.

Types of Slots Editors

There are many different types of slot editors. The type of slot editor is dependent on the type of slot you are editing. In the new slot dialog, there is a popup menu where you can specify the type of editor. Many of the predefined slots also have specific types of editors that are worth a brief explanation.

viewEffect

You can set various view effects in the effect picker in this editor (see Figure G.39).

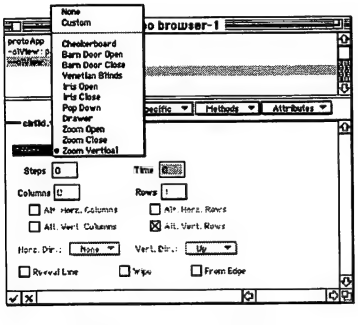


Figure G.39 The viewEffect slot.

viewFormat

You can set various aspects of a view's frame and fill in this editor (see Figure G.40).

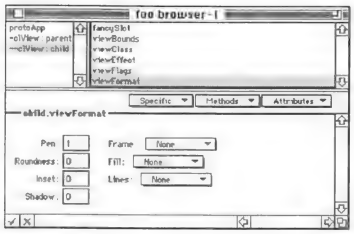


Figure G.40 The viewFormat slot.

viewBounds

You can set the left, right, top, and bottom bounds of a view template in this editor (see Figure G.41).

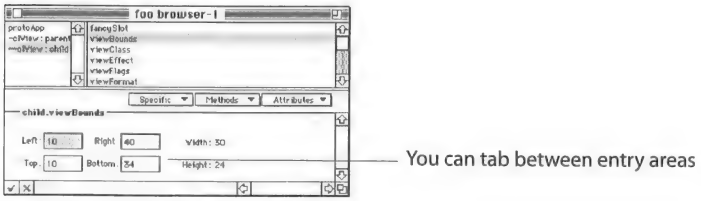


Figure G.41 The viewBounds slot.

viewFlags

You can set various view options in this editor (see Figure G.42).

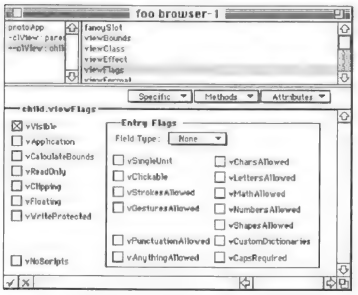


Figure G.42 The viewFlags slot.

The viewJustify Slot Editor

You set the justification of one view relative to another in this editor. Note that you can also set the internal justification of graphics and text (see Figure G.43).

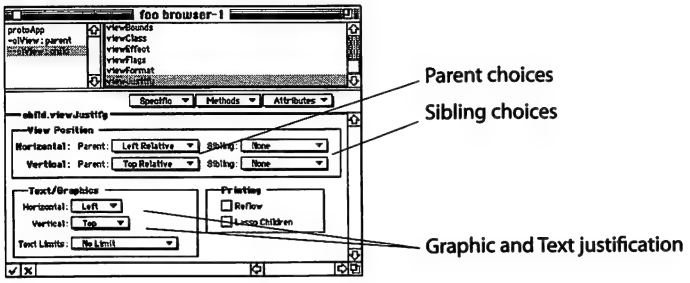


Figure G.43 The viewJustify slot.

Additional Parts of Your Project

The Project Data File

The "Project Data" file is a text file in which you can define the following:

constants	These are constants available anywhere in your application. They might include the application symbol, for instance.
InstallScript	A function executed as your package is installed.
RemoveScript	A function executed when your package is deinstalled. Note that your application templates are no longer present when this function is called.

You can edit this file in any text editor, or you can edit it within NTK by selecting Project Data from the Project menu (which creates the file in your project folder if it doesn't already exist).

Project Settings

The Project Settings dialog (see Figure G.44) controls various aspects of the project build. Select Settings from the Project menu to bring up the dialog.

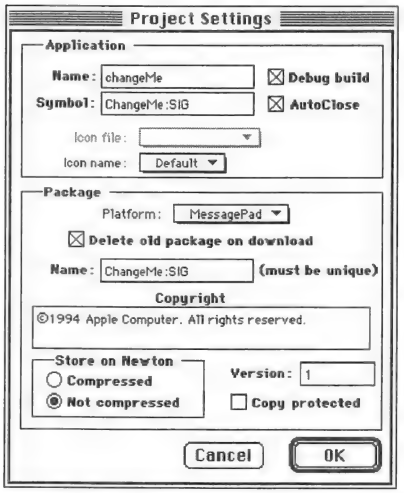


Figure G.44 Project Settings dialog.

Name

This is the name of the application that appears in the Extras drawer.

Symbol

This is the unique application symbol. For more information on creating this symbol, see "Creating Unique Application Symbols and Names" on page 344. At run time, a slot is created in the root view with this symbol; the slot's value is the application base view.

Debug Build

If this option is set, then NTK creates a `debug` slot for all named templates (if you have manually added such a slot to a template, NTK does not override it). The value stored in the slot is the name of the template. The `debug` slot is created when the project is compiled, therefore you do not see it in the browser.

In addition, NTK sets the compile-time constant `debugOn` to true if Debug Build is checked. If the option is off, the constant is set to `NIL`. For details on using this constant in your code, see "Remove Debug Code for Non-Debug Builds" on page 288.

Auto-Close

If this is checked, opening this application closes all other auto-close applications. You should normally keep this checked.

Icon File/Icon Name

These two popups control the icon that displays above your application name in the Extras drawer.

The Icon File popup presents a list of resource files in your project. The Icon Name popup presents a list of named PICT resources from the chosen file.

Platform

Choose a platform (one of the files in the Platforms folder). The platform determines the size of the drawing view shown in the layout window.

Package Name

This unique name appears when the user selects Remove Software.

Copyright

This string is embedded in your package.

Store on Newton

These following two options have no effect on how NTK builds your application. They only control how the package is stored on the Newton:

Compressed	When selected, the package is compressed on the Newton. It will be smaller but slower.

Not Compressed	When selected, the application is not compressed.

Version

This number is used to differentiate between different versions of the same application.

Copy Protected

Setting this flag specifies that the application should not be copied. In particular, this means that Newton Connection will not copy the application when doing a backup. Other software that copies applications should respect this flag, although there is nothing that requires it.

Building and Downloading

Building a Package

To build an application from the project file, use Build Package from the Project menu. This builds a package file in the same folder as your project. It uses the name of the project followed by the suffix ".pkg".

NTK Toolkit App

The NTK Toolkit App on the Newton is used for downloading packages (including applications) you build with NTK. It is also used to debug your applications using the Inspector.

The Inspector

NTK contains an Inspector window, where you type code that is compiled into NewtonScript, downloaded to the Newton, executed, and then the result is displayed in the Inspector window. For more information on using the Inspector, see "The Inspector" on page 268.

In order to work, the Inspector must be connected to the Newton. Here is how to connect the Inspector:

On the Macintosh

- Select Connect Inspector from the Window menu.

On the Newton

- Tap the Toolkit icon to open the Toolkit App slip (make sure the type of connection is set to Macintosh serial).
- Tap the Connect Inspector button. The Newton displays a progress slip momentarily and then that slip disappears (see Figure G.45).

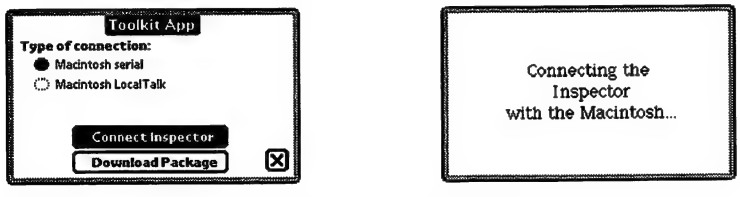


Figure G.45 Connecting the Inspector on the Newton.

Selecting Open Inspector on the Macintosh merely opens the Inspector window. You still need to select Connect Inspector to establish a connection.

Downloading a Package

You must already have installed NTK Toolkit App and have a serial cable between your Newton and your Macintosh (see "Installing NTK" on page 348) in order to download a package. Choose Download Package from the Project menu. Then, on your Newton tap the Toolkit icon to open the Toolkit App slip. Make sure the type of connection is set to Macintosh serial and tap the Download Package button. The Newton will display a progress slip, while NTK will display a progress dialog. Within a few seconds, your package should appear in the Extras drawer.

Downloading a Package with the Inspector Connected

While the Inspector is connected, you can download packages automatically from the Macintosh—just select Download Application from the Project menu.

Index

A

B

C

D

E

F

G

H

I

J

O

P

Q

R

S

T

U

V

W

Companion Disk Order Form

The complete source code from this book and more is available on the companion disk to this volume. This disk includes:

- A complete project for each stage of development (for every point a build and download are suggested).
- An extended WaiterHelper project with support for the Action button (Print, Fax, Mail, Beam, Duplicate, Delete, and Move To/From Card).
- A demo version of Jason Harper's ViewFrame, the Newton-based debugging tool.

Price: $24.95

Shipping and handling: US add $3, International add $10. California residents, please add sales tax. VISA/MC/Checks accepted. Sorry, no purchase orders.

Send orders to: Calliope Enterprises, Inc.,
700 East Redlands, Blvd., Suite 154
Redlands, CA 92373

Or, call *(800) 839-9699* or *(909) 793-2545*, to order with credit card.

Name _____

Address _____

City/State/Zip _____ Phone _____

Credit Card Type and # _____

Signature _____ Expiration Date _____

Although we strive to fill each order promptly, please allow 4 to 6 weeks for delivery. All sales are final.

~TO ORDER NEWTON TOOLKIT~
CALL APDA AT 1-800-282-2732

FOR INFORMATION ON DEVELOPING FOR NEWTON
CALL NEWTON DEVELOPER RELATIONS AT 408-862-7226

FOR NEWTON MESSAGEPAD SUPPORT
CALL 1-800-SOS-APPL

NEWTON DEVELOPER RELATIONS
APPLE COMPUTER, INC.
1 INFINITE LOOP
MS 305-3A
CUPERTINO, CA 95014

~FOR NEWTON TRAINING~

APPLE DEVELOPER UNIVERSITY
20525 MARIANI AVENUE, M/S 305-1TU
CUPERTINO, CA 95014
TELEPHONE: 408-974-4897
FACSIMILE: 408-974-0544
APPLELINK: DEVUNIV
INTERNET:DEVUNIV@APPLELINK.APPLE.COM

Julie McKeehan / Neil Rhodes

PROGRAMMING FOR THE NEWTON®

Software Development with NewtonScript™

This book is an indispensable tool for Newton® programmers. Readers will learn how to develop software for the Newton®, on the Macintosh®, from the people that developed the course on programming the Newton® for Apple Computer. The enclosed floppy disk provides sample code for examples in the book, as well as a demonstration version of Newton Toolkit™ (NTK™), Apple Computer's complete development environment for the Newton®.

- **Hands-on Newton® development training** with a sample Newton® application built from the ground up.
- **Authors are external faculty at Apple Developer University**, teaching classes on programming the Newton®.
- Enclosed floppy disk contains source code for a Newton® application, as well as **Demonstration NTK™**.
- Assumes programming experience, but not in any particular language.
- Teaches key concepts of **object-oriented programming** to facilitate programming with NewtonScript™.
- Foreword by NewtonScript™ principal designer, Walter R. Smith.

About the Authors

Julie McKeehan and Neil Rhodes are external faculty at Apple Developer University and principals of Calliope Enterprises, a company providing Macintosh® and Newton® programming and training services to software developers. Julie and Neil are also the authors of the best-selling book, *Symantec C++ Programming for the Macintosh*.

ISBN 0-12-484800-1
90056
9 780124 848009